Introduction

This book provides guidelines for developing multimedia subsystems. Each subsystem component is described in detail in individual chapters. Models are used to complement the information provided by component sample program templates.

Additional Multimedia Information

Multimedia REXX - (online)
Describes REXX functions that enable media control interface string commands to be sent from an OS/2 command file to control multimedia devices. This online book is provided with OS/2 multimedia.

Guide to Multimedia User Interface Design - (41G2922)
Describes design concepts to be considered when designing a CUA multimedia interface that is consistent within a particular multimedia product and across other products.

OS/2 Multimedia Device Driver Developers: The *Device Driver Source Kit for OS/2* contains documented source code and detailed information on how to build a virtual device driver (VDD) and audio physical device driver (PDD) for OS/2 multimedia.

The kit covers the complete range of OS/2 physical and virtual device drivers, from printers, displays, SCSIs and CD-ROM drives, to the device drivers for Pen for OS/2 and OS/2 multimedia. Comprehensive descriptions of all the device driver interfaces and system services are included.

Using This Online Book

Before you begin to use this online book, it would be helpful to understand how you can:

- Expand the Contents to see all available topics
- Obtain additional information for a highlighted word or phrase
- Use action bar choices.

How To Use the Contents

When the Contents window first appears, some topics have a plus (+) sign beside them. The plus sign indicates that additional topics are available.

To expand the Contents if you are using a mouse, select the plus sign (+). If you are using a keyboard, use the Up or Down Arrow key to highlight the topic, and press the plus key (+).

To view a topic, double-click on the topic (or press the Up or Down Arrow key to highlight the topic, and then press Enter).

How To Obtain Additional Information

After you select a topic, the information for that topic appears in a window. Highlighted words or phrases indicate that additional information is available. You will notice that certain words in the following paragraph are highlighted in green letters, or in white letters on a black background. These are called hypertext terms. If you are using a mouse, double-click on the highlighted word. If you are using a keyboard, press the Tab key to move to the highlighted word, and then press the Enter key. Additional information will appear in a window.

How To Use Action Bar Choices

Several choices are available for managing information presented in the M-Control Program/2 Programming Reference. There are three pull-down menus on the action bar: the **Services** menu, the **Options** menu, and the **Help** menu.

The actions that are selectable from the **Services** menu operate on the active window currently displayed on the screen. These actions include the following:

Bookmark
Sets a place holder so you can retrieve information of interest to you.

When you place a bookmark on a topic, it is added to a list of bookmarks you have previously set. You can view the list, and you can

remove one or all bookmarks from the list. If you have not set any bookmarks, the list is empty.

To set a bookmark, do the following:

1. Select a topic from the Contents.
2. When that topic appears, choose the **Bookmark** option from the **Services** menu.
3. If you want to change the name used for the bookmark, type the new name in the field.
4. Select the **Place** radio button (or press the Up or Down Arrow key to select it).
5. Select **OK**. The bookmark is then added to the bookmark list.

Search

Finds occurrences of a word or phrase in the current topic, selected topics, or all topics.

You can specify a word or phrase to be searched. You can also limit the search to a set of topics by first marking the topics in the Contents list.

To search for a word or phrase in all topics, do the following:

1. Choose the **Search** option from the **Services** pull-down.
2. Type the word or words to be searched.
3. Select **All sections**.
4. Select **Search** to begin the search.
5. The list of topics where the word or phrase appears is displayed.

Print

Prints one or more topics. You can also print a set of topics by first marking the topics in the Contents list.

You can print one or more topics. You can also print a set of topics by first marking the topics on the Contents list.

To print the document Contents list, do the following:

1. Select **Print** from the **Services** menu.
2. Select **Contents**.
3. Select **Print**.
4. The Contents list is printed on your printer.

Copy

Copies a topic you are viewing to a file you can edit.

You can copy a topic you are viewing into a temporary file named TEXT.TMP. You can later edit that file by using an editor such as the System Editor.

To copy a topic, do the following:

1. Expand the Contents list and select a topic.
2. When the topic appears, select **Copy to file** from the **Services** menu.

The system copies the text pertaining to that topic into the temporary TEXT.TMP file.

For information on any of the other choices in the **Services** menu, highlight the choice and press the F1 key.

Options

Changes the way the Contents is displayed.

You can control the appearance of the Contents list.

To expand the Contents and show all levels for all topics, select **Expand all** from the **Options** menu.

For information on any of the other choices in the **Options** menu, highlight the choice and press the F1 key.

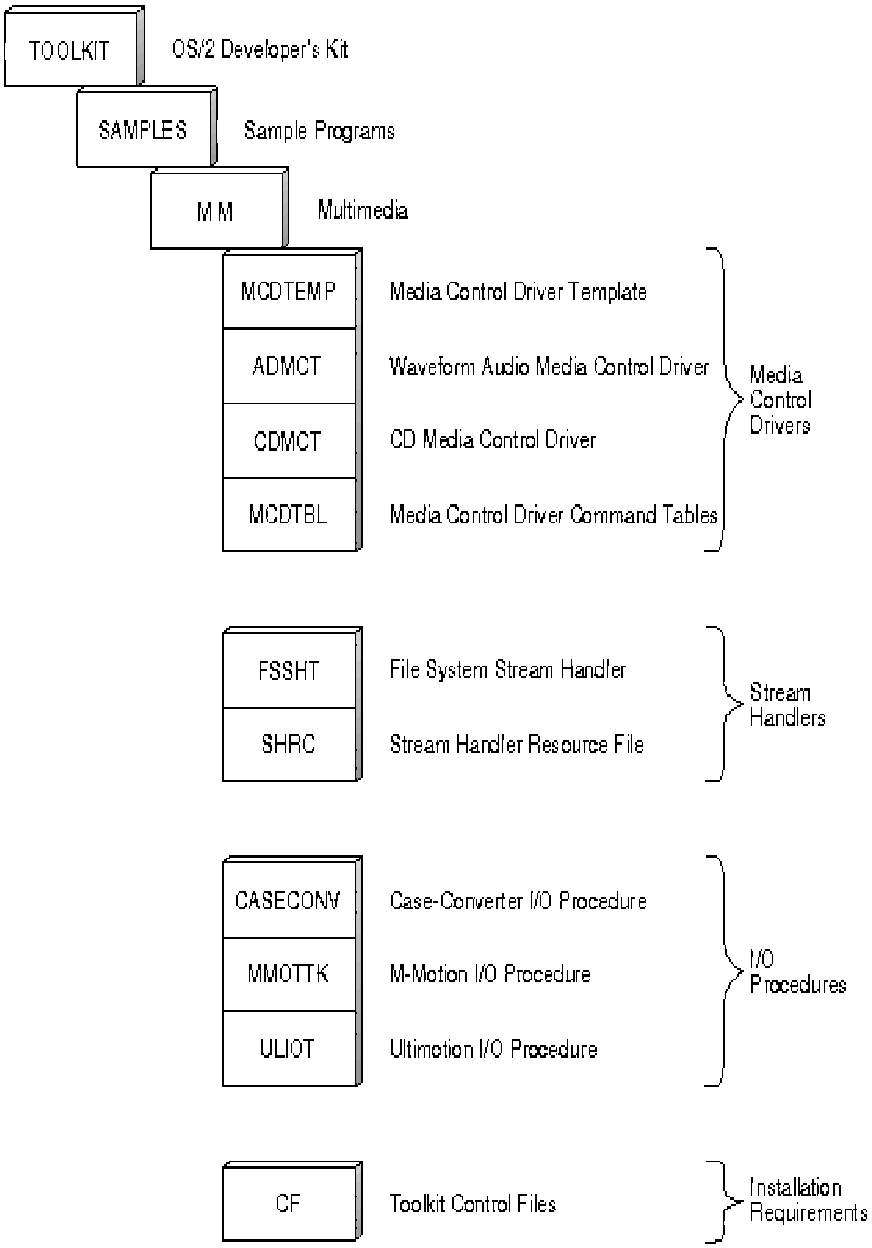
What's New...

This release of the *Multimedia Subsystem Programming Guide* includes the following addition:

- Real-Time MIDI Subsystem

Multimedia Subsystems Overview

OS/2 multimedia (referred to as Multimedia Presentation Manager/2 or MMPM/2 in previous releases) has an extendable architecture that makes it possible to add new functions, devices, and multimedia data formats as the technology of multimedia advances. This chapter provides a general overview of the subsystem components provided with the OS/2 multimedia system. Subsequent chapters include detailed guidelines on how to develop and install your own OS/2 multimedia subsystem through the use of the sample programs illustrated in the following figure. Each sample program serves as a template that can be modified easily to meet your multimedia requirements. Use these sample programs to develop and install media control drivers, stream handlers, and I/O procedures for OS/2 multimedia.



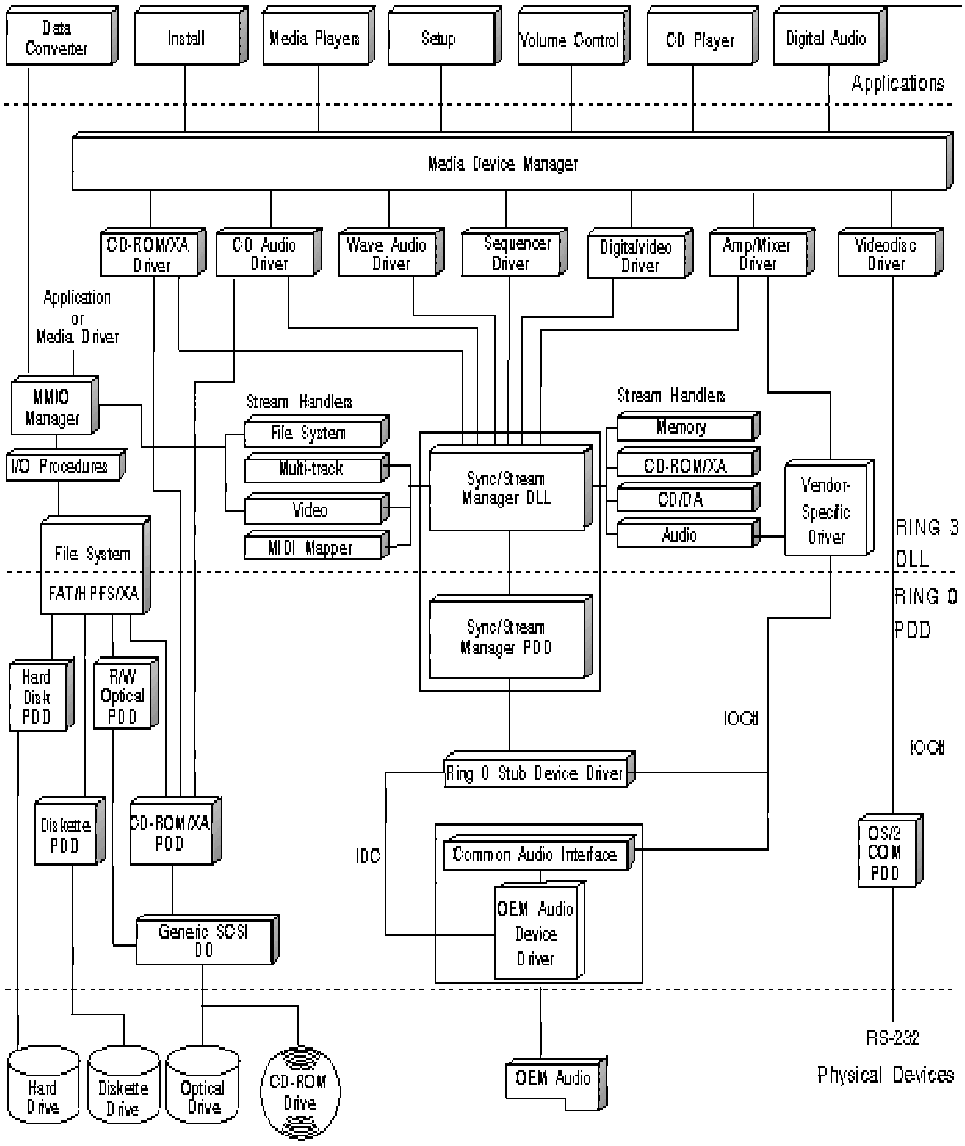
OS/2 Multimedia System Architecture

The following figure illustrates the subsystem components provided by the OS/2 multimedia system. These subsystems (media control drivers, stream handlers, and I/O procedures) are controlled by *managers* that oversee a range of activities in the OS/2 multimedia environment.

At the Ring 3 level, OS/2 multimedia employs a *Media Device Manager (MDM)* , which manages logical media devices representing audio adapters, CD-ROM drives, and other hardware devices. Amongst its duties, MDM determines which process gains access when two or more applications attempt to control a media device.

The *Sync/Stream Manager (SSM)* is also available to manage streaming and synchronization calls initiated by the media control drivers. This eliminates the need for each media driver to provide its own solution for these common multimedia requirements. Pairs of stream handlers implement the transport of data from a source to a target device while the SSM provides coordination and central management of data buffers and synchronization data.

Lastly, the *MMIO Manager* enables subsystem components such as media control drivers and applications to access and manipulate a variety of data objects, including images, graphics, digital audio, and digital video. These objects can be stored in a variety of file formats on a variety of storage systems. The MMIO Manager uses installable I/O procedures to direct the input and output associated with reading from and writing to different types of storage systems or file formats.



Extendable Device Support

The system architecture of OS/2 multimedia extensions is designed to be extendable. This level of modularity allows independent development of support for new hardware devices, logical media devices, and file formats.

Examples of media control interface devices are listed in the following table. The purpose of the table is to show the logical device types that can be supported and already have media control interface definitions. Devices currently supported by OS/2 multimedia are indicated by (X) marks.

Media Device Type	OS/2 Multimedia	String	Constant

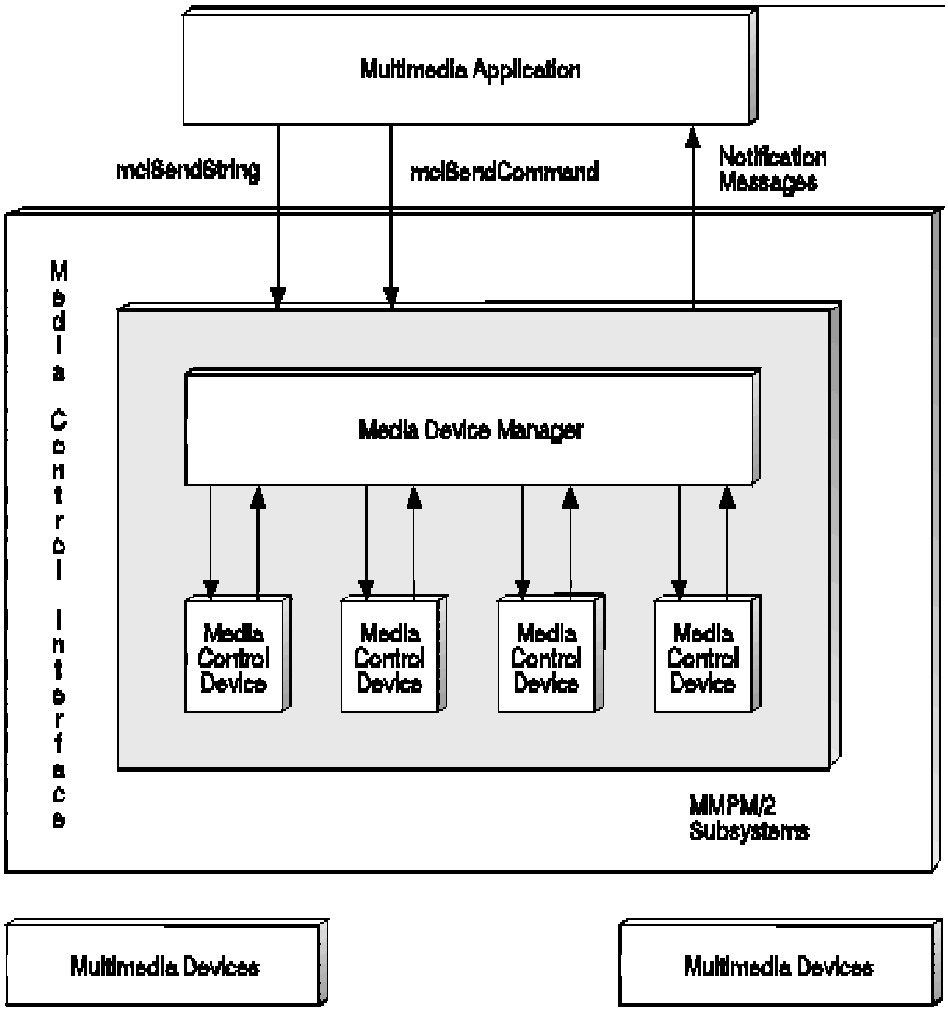
Amplifier mixer	X	ampmix	MCI_DEVTYPE_AUDIO_AMPMIX
Audio tape player		audiotape	MCI_DEVTYPE_AUDIO_TAPE
CD audio player	X	cdaudio	MCI_DEVTYPE_CD_AUDIO
CD-XA player	X	cdxa	MCI_DEVTYPE_CDXA
Digital audio tape		dat	MCI_DEVTYPE_DAT
Digital video player	X	digitalvideo	MCI_DEVTYPE_DIGITAL_VIDEO
Headphone		headphone	MCI_DEVTYPE_HEADPHONE
Microphone		microphone	MCI_DEVTYPE_MICROPHONE
Monitor		monitor	MCI_DEVTYPE_MONITOR
Other		other	MCI_DEVTYPE_OTHER
Video overlay		videooverlay	MCI_DEVTYPE_OVERLAY
Sequencer	X	sequencer	MCI_DEVTYPE_SEQUENCER
Speaker		speaker	MCI_DEVTYPE_SPEAKER
Videodisc player	X	videodisc	MCI_DEVTYPE_VIDEODISC
Video tape/cassette		videotape	MCI_DEVTYPE_VIDEOTAPE
Waveform audio player	X	waveaudio	MCI_DEVTYPE_WAVEFORM_AUDIO

Note: M-Control Program Version 2.01, which supports the M-Motion Video Adapter/A, provides overlay extensions for OS/2 multimedia.

Media Control Drivers

The media control interface provides the primary mechanism for application control of media devices. The top layer consists of the Media Device Manager (MDM), which provides resource management for media devices. The bottom layer consists of *media control drivers (MCDs)* -dynamic link libraries that implement the function of a media device.

Applications interact with the media control interface (and thus with media devices) in two ways, through either a procedural interface (mciSendCommand) or a string interface (mciSendString). However, before an MCD can interpret a string command, the MDM must use a command table to change the string into an equivalent procedural command.



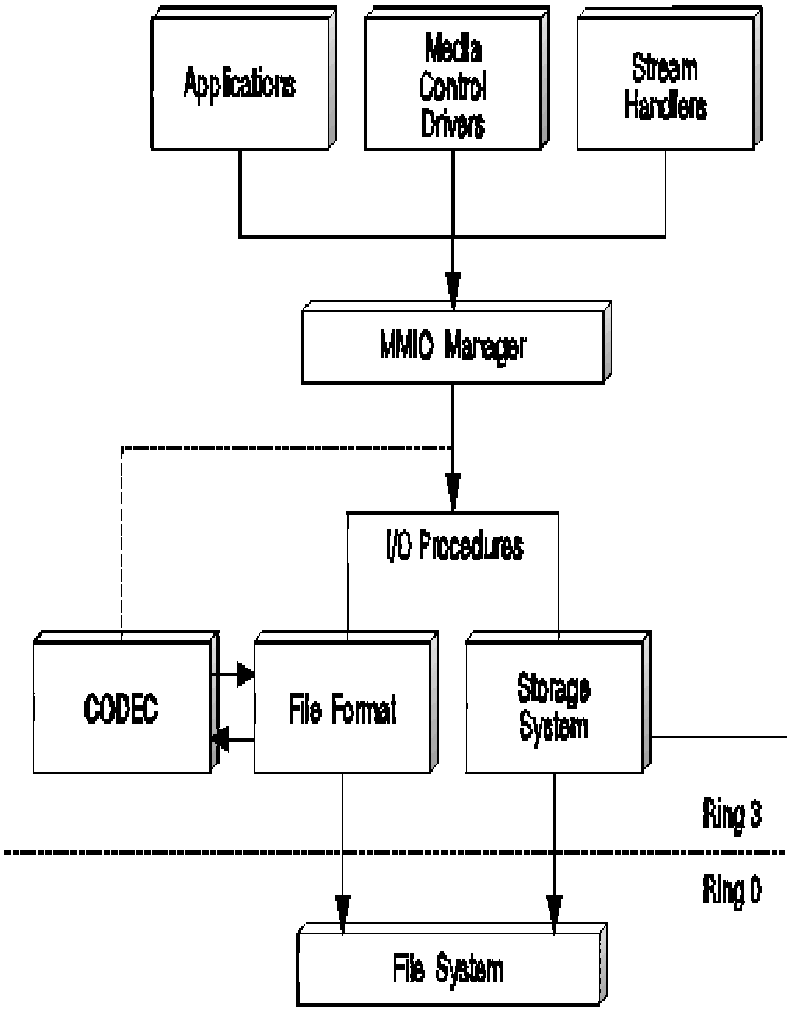
MCDs do not directly control hardware devices. Instead, they pass commands through a subsystem or physical device driver interface. This arrangement frees MCDs from having to be knowledgeable about the hardware in order to implement its function. However, MCDs have to be knowledgeable about the *means* of implementing that function. For example, a CD-Audio player that is implemented using a CD-ROM drive with a built-in Digital-to-Analog Converter (DAC) can perform its function by simply issuing device IOCtl commands to the drive's device driver. However, a CD-Audio player that uses a separate Digital Signal Processor (DSP) for playback of the digital audio data is implemented quite differently, calling functions to manage a DSP coprocessor and data transfer between the drive and the coprocessor.

MCDs can also use the services of the other OS/2 multimedia subsystems such as the Stream Programming Interface (SPI). This subsystem provides data streaming services that allow stream-handlers to control the flow of data from one device to another in real time, maintaining a continuous flow of data between physical devices.

I/O Procedures

Multimedia input/output (MMIO) services provides both I/O and CODEC procedures. I/O procedures are message-based handlers, which direct the input and output associated with reading and writing to different types of storage systems or file formats. Applications and the MMIO subsystem communicate to I/O procedures (DLL files) through the use of MMIO messages. When MMIO receives a request from an application through a function call, the MMIO Manager sends a predefined message for that operation to the I/O procedure responsible for that particular file format or storage system. In turn, the I/O procedure performs operations based on the messages it receives from the MMIO Manager or an application.

These messages are designed for efficient communications to all I/O procedures. The I/O procedures, however, must be able to process the messages or pass them on to a child I/O procedure. For example, if an I/O procedure receives a message requesting the compression of data object, the I/O procedure must be able to process the message, or pass the message to a CODEC procedure. The following figure illustrates the interaction of I/O and CODEC procedures in the MMIO subsystem.



The MMIO Manager calls the following types of procedures:

File Format

A file format procedure is an I/O procedure that manipulates data at the element level, with each procedure handling a different element type such as audio, image, or MIDI. It processes the data "within" the object and does not rely on any other file format I/O procedures to process the data. However, a file format procedure might need to call a storage system I/O procedure to obtain data within a file containing multiple elements.

Storage System

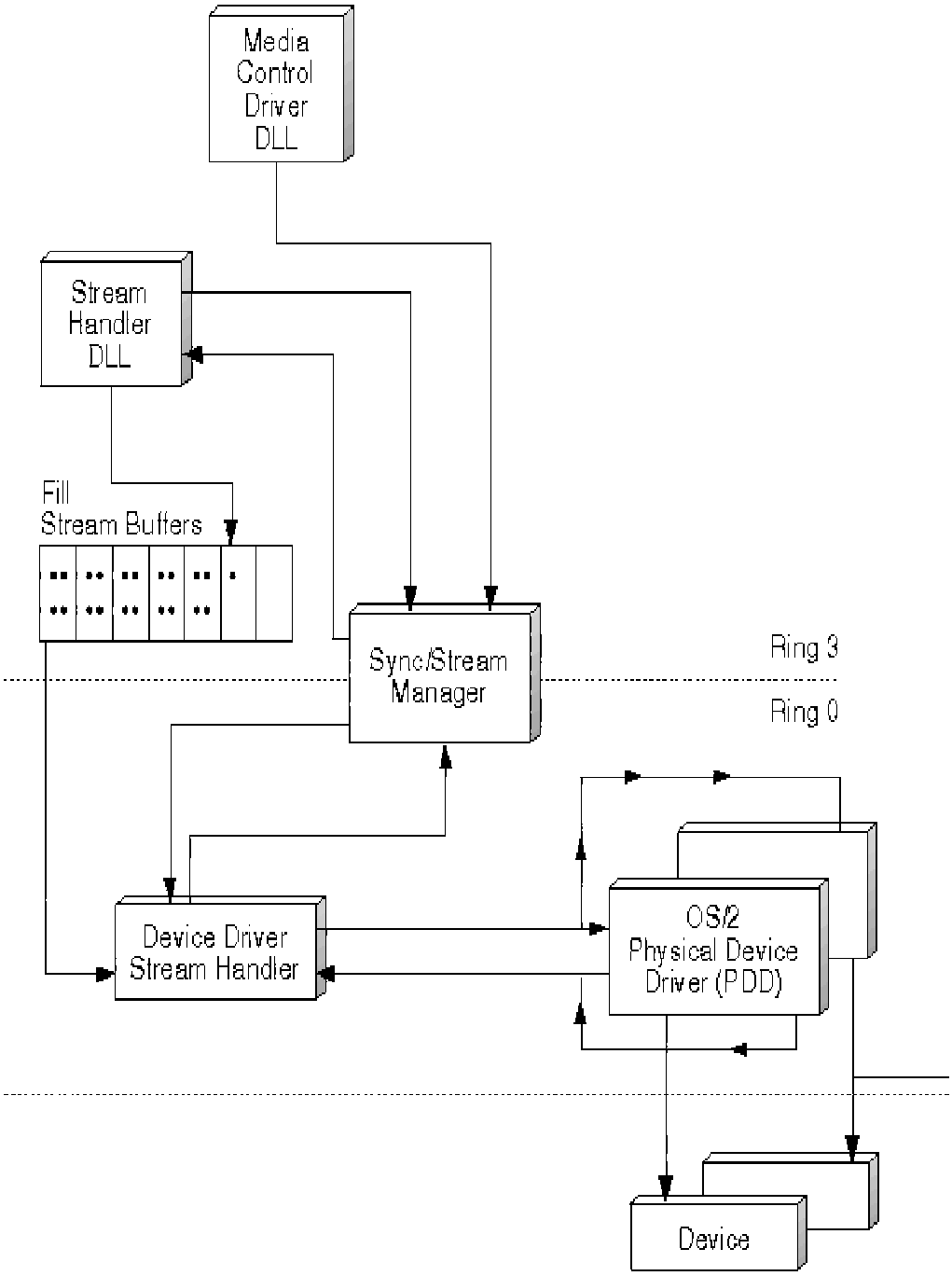
A storage system procedure is an I/O procedure that "unwraps" a data object for a file format procedure to access. Storage system procedures are unaware of the format of the data object contained within the wrapper.

CODEC

A CODEC procedure operates on data within a file or buffer. Based on the data content, an I/O procedure can load a CODEC procedure to either compress or decompress data.

Stream Handlers

The multimedia system provides stream handlers at both the system kernel level (Ring 0) and the application level (Ring 3). Stream handlers are at Ring 0 and Ring 3 because some streams are ideally controlled by a direct connection between the stream handler and a device's physical device driver (PDD). Other streams are not associated with a data source or target, which maps physically to a specific device. For example, the file system stream handler is a DLL, because all file system input/output (I/O) functions are available as Ring 3 OS/2 functions, and service all file system devices. This eliminates the need to build a specific stream handler device driver for every device the file system can access.



Stream handlers are responsible for controlling the flow of application data, in a continuous, real-time manner. Each handler can establish multiple data stream instances, where each stream involves data of a specific type; for example, MIDI (Musical Instrument Digital Interface) or ADPCM (Adaptive Delta Pulse Code Modulation). The application, through the use of a media control driver, invokes an SPI function to create the stream and another SPI function to activate the data stream. The application does not have to continuously invoke SPI functions to maintain data flow. Instead, the stream handler keeps the I/O continuous, simplifying the operations of the application.

Media Control Drivers

This section shows you how to write a media control driver (MCD) by illustrating the programming interfaces used by streaming and nonstreaming MCDs. The following discussion focuses on the MCDs depicted in the Duet Player sample programs (see the *OS/2 Multimedia Application Programming Guide* for details.) The waveform audio MCD uses SPI data streaming services to handle the creation and management of source and target stream handlers. However, the CD audio MCD typically has no data flow associated with it as most

CD audio devices process data internally (internal to the device without any help from the main CPU).

Source code is provided for the following media control driver samples located in the \TOOLKIT\SAMPLES\MM subdirectory:

Media Control Driver Template (MCDTEMP)
: Provides a basic template to write an MCD. Refer to the ADMCT and CDMCIDRV subdirectories for specific streaming or MMIO samples.

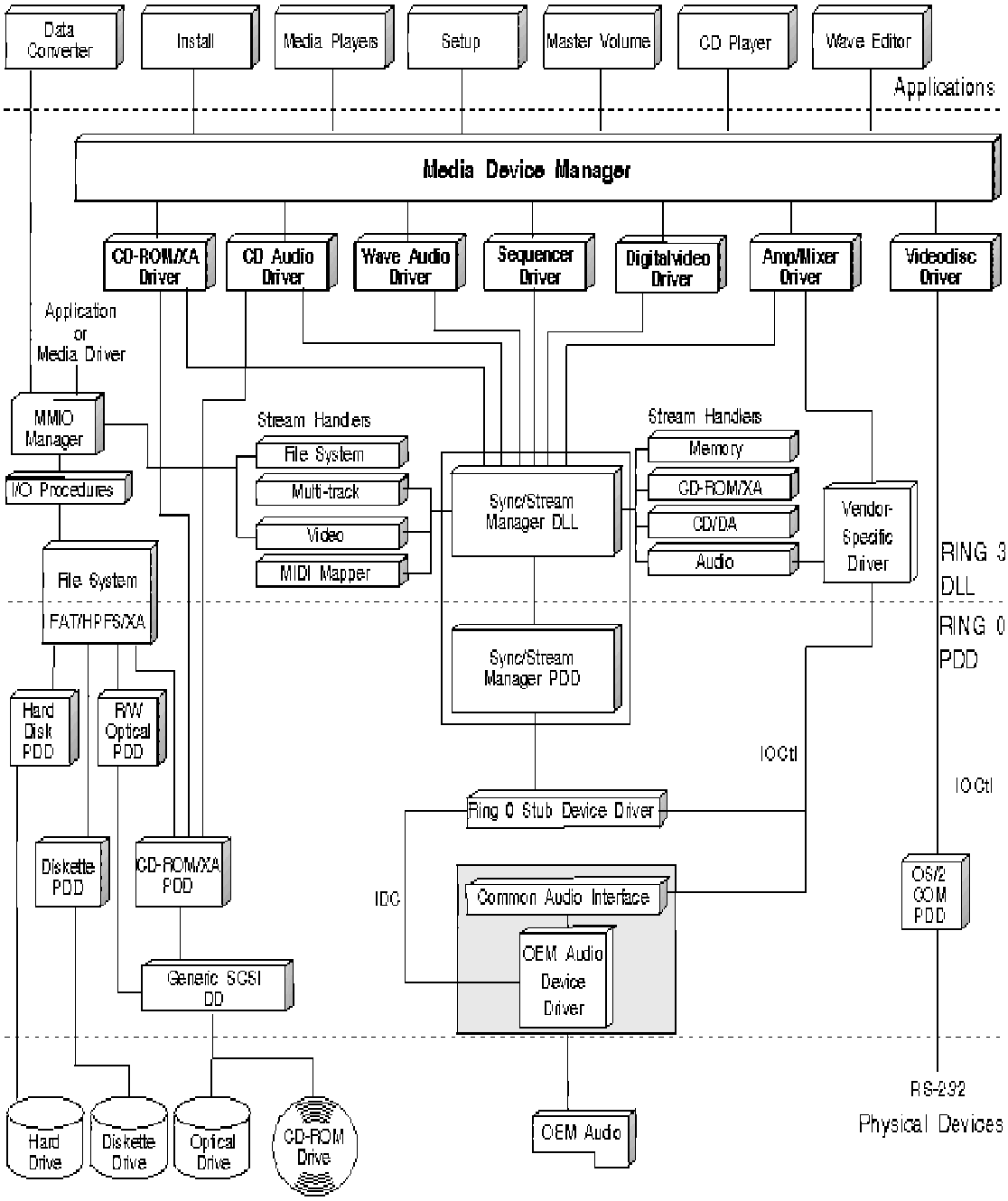
Waveform Audio Media Control Driver (ADMCT)
: Provides an example of how to control a *streaming* device. *Streaming* devices use the services of the Sync/Stream Manager (SSM) of OS/2 multimedia to control the data stream from a source location to a target location. For example, the audio adapter used in Duet Player I is a streaming device which plays back waveform audio files stored on the user's hard disk. The PM application sends a request to the Media Device Manager (MDM) to play the file. In turn, the MDM calls the waveform audio MCD, which uses stream handlers to buffer data from the waveform file on disk (source) to the audio adapter (target).

CD Media Control Driver (CDMCIDRV)
: Provides an example of how to control a *nonstreaming* device. *Nonstreaming* devices stream data within the device. A device that streams data internally does not need to use buffered I/O because the source and destination of the data is within the device. Therefore, a nonstreaming device does not require the use of the SSM subsystem of OS/2 multimedia. For example, the CD-ROM device used in Duet Player II is a nonstreaming device. When the CD audio MCD receives a PLAY, PAUSE, or STOP command from the program, it issues the appropriate IOCtls to the CD-ROM device to perform the function. The hardware does all the work, relaying the digital information off the disc, translating it into an audio signal, and piping it to a port such as a headphone jack.

Media Control Driver Architecture

The objects that receive the device control commands issued by multimedia applications using mciSendCommand or mciSendString are called media control drivers. Media control drivers are OS/2 dynamic link libraries (DLLs) that provide a level of hardware independence for applications. The Media Device Manager (MDM) provides the interface to these media objects. The MDM also provides a level of component management, making it possible for an application to synchronize its use of MCDs as well as share MCDs with other applications.

The following figure illustrates the MCDs provided with the OS/2 multimedia system.



Media Control Driver Entry Point

All MCDs have commands passed to them in the same way, regardless of whether they support streaming or nonstreaming devices. Applications call either the mciSendCommand or mciSendString functions to pass commands to the entry point, mciDriverEntry. In addition, some commands are generated by the MDM itself. These are the commands for saving and restoring an instance.

The following table shows the parameters associated with mciDriverEntry.

Parameters	Description
PVOID *pInstance*	Pointer to instance structure of driver.

USHORT *usMessage*	The requested action to be performed.
ULONG *ulParam1*	Flag for the message. This flag is defined separately for each message.
PVOID *pParam2*	Second data parameter, whose interpretation is dependent on the message.
USHORT *usUserParm*	User parameter returned on notification message.

The function of mciDriverEntry is to switch, based on the message, and perform the appropriate task. An example of a message is MCI_OPEN.

Your driver must be able to handle messages in the following fashion:

1. Your driver must handle all the required messages.
2. Your driver must handle device-type messages for its particular class of device.

   For example, if you are writing an MCD for a videodisc player, you must parse the videodisc-specific messages.
3. Your driver must handle messages that are specific to the device your driver supports. Suppose your driver controls a device that is one of the following device types:

   CD-ROM/XA
   CD Audio
   Wave Audio
   Sequencer
   Digital Video
   Amp Mixer
   Videodisc

   MDM has already defined device-type messages for these devices, which means a command table exists for each of these device types. If you want to support device-specific messages, you must create a device-specific command table.

If your driver is for a device type other than the device types listed, you must create a command table that includes both the device-type messages and the device-specific messages.

Types of Command Messages

The Media Device Interface is a set of defined and extendable media control commands. How the MCD communicates with the appropriate hardware device drivers to perform the requested command message is completely up to the MCD. Device commands used by MCDs are grouped into required, basic, and system command messages.

Note: Refer to the *OS/2 Multimedia Programming Reference* for syntax and associated parameters.

Required Command Messages

Required commands are recognized by *all* devices and have actions common to all media devices. The following table lists the required commands that must be supported by your MCD.

Message	Description

MCI_CLOSE	Closes the device.
MCI_GETDEVCAPS	Gets the capabilities of a device.
MCI_INFO	Gets textual information from the device.
MCI_OPEN	Initializes an instance of the device.
MCI_STATUS	Gets status information from the device.
MCIDRV_SAVE	Is sent from MDM to MCDs to save the context.
MCIDRV_RESTORE	Is sent from MDM to MCDs to restore the state of an inactive device context.

The required command messages use a ULONG for the *ulParam1* parameter that contains any flags for the command message. They also use the *pParam2* parameter for a pointer to a message-specific data structure. Your MCD might create extended commands by adding new flags and data structure fields to those already defined. When you extend a command message, your MCD must still support the required flags and fields.

The following table identifies the flags and data structures of the required command messages. For a complete reference of media control interface commands, refer to the *OS/2 Multimedia Programming Reference* .

Message	Parameters (ulParam1)	Data Structure (pParam2)
MCI_CLOSE	MCI_NOTIFY MCI_WAIT	MCI_GENERIC_PARMS
MCI_GETDEVCAPS	MCI_NOTIFY MCI_WAIT MCI_STATUS MCI_GETDEVCAPS_EXTENDED MCI_GETDEVCAPS_MESSAGE MCI_GETDEVCAPS_ITEM	MCI_GETDEVCAPS_PARMS
MCI_INFO	MCI_NOTIFY MCI_WAIT MCI_INFO_PRODUCT	MCI_INFO_PARMS
MCI_OPEN	MCI_WAIT MCI_OPEN_SHARABLE MCI_OPEN_ELEMENT MCI_OPEN_MMIO	MMDRV_OPEN_PARMS
MCI_STATUS	MCI_NOTIFY MCI_WAIT MCI_STATUS_CLIPBOARD MCI_TRACK MCI_STATUS_ITEM	MCI_STATUS_PARMS
MCIDRV_RESTORE	MCI_WAIT MCI_SHAREABLE MCI_EXCLUSIVE	MCI_GENERIC_PARMS
MCIDRV_SAVE	MCI_WAIT	MCI_GENERIC_PARMS

Opening an MCD

MCI_OPEN is the first message received by the MCD. This message instructs the driver to create and initialize an instance of a particular device. The MCD must allocate and initialize the instance structure. Note that the MCI_OPEN message does not make the instances active.

Because the MDM needs to pass additional information to the drivers, the open structure for MCDs is different from the MCI_OPEN_PARMS structure. In addition, the MCDs need to return information to the MDM. If the application requests a NOTIFY on the

MCI_OPEN message, the MCD sends back the open NOTIFY on the MCIDRV_RESTORE message. This is transparent to the application. The MCD does not receive any of the following flags on an MCI_OPEN:

- MCI_OPEN_ALIAS
- MCI_NOTIFY
- MCI_OPEN_TYPE_ID

On the MCI_OPEN message, *pParam2* points to the MMDRV_OPEN_PARM structure located in the MMDRVOS2.H file. This structure contains information for the MCD.

Field	In/Out	Description
HWND hwndCallback	In	Window handle used for mciDriverNotify.
USHORT usDeviceID	In	Device ID assigned to this instance. This field is filled in by MDM.
USHORT usDeviceType	In	Device type number for this instance.
USHORT usDeviceOrd	In	Device ordinal number for this instance.
PVOID pInstance	InOut	Pointer to instance structure initialized by driver. The driver fills in this parameter.
CHAR szDevDLLName[260]	In	Character string containing device-specific DLL name to call for the open.
PSZ pszElementName	In	Typically a file name. If OPEN_PLAYLIST is specified, this is a pointer to a memory playlist. If OPEN_MMIO is specified, this is a MMIO handle.
USHORT usDevParmLen	In	Device parameters data block length.
PVOID pDevParm	In	Device parameters data block. This data is unique to each type of device and is retrieved from the MMPM2.INI file. (for example, LVD "PORT=COM1 SPEED=9600N71").
USHORT usResourceUnitsRequired	InOut	Number of resource units this instance requires. See Resource Units and Classes.
USHORT usResourceClass	InOut	Resource class this instance belongs to. See Resource Units and Classes.
USHORT usResourcePriority	InOut	Resource priority for this instance.
ULONG ulParam2	In	Pointer to MCI_OPEN structure.

Subsystem Messages

MDM provides resource management to MCDs using the following messages:

- MCIDRV_SAVE
- MCIDRV_RESTORE

MCIDRV_RESTORE and MCIDRV_SAVE allow the MDM to tell the MCD when to make a device context active and inactive. These commands are always sent with the MCI_WAIT flag on. Not all MCDs need to support these messages completely, but they are provided. **The MCIDRV_SAVE and MCIDRV_RESTORE messages are not sent by applications.**

After sending an MCI_OPEN message, the MDM sends an MCIDRV_RESTORE message to the MCD to make the device context active. The MCD should *not* expect to always receive an MCIDRV_RESTORE message right after an MCI_OPEN. Certain conditions require a device context to be opened but not made active. One such condition is if a device context is opened as non-shareable, and then a second device context is opened as shareable. The second device context is opened and put in the inactive stack by the MDM. Therefore, MCDs

should not make a device context active on an open, but only on an MCIDRV_RESTORE.

MDM sends an MCIDRV_SAVE message to the MCD to make an instance inactive. When the MCD receives the save message, it should query the state of the device (its file location, track, volume level, and so on) and save this data in the instance structure. The instance should also be placed in a paused state. Upon receiving the restore command, the driver should restore the device based on the state information in the instance. If the device was playing or recording when it was saved, then it should be put back into that state on the restore.

The following commands can be sent to inactive instances:

- MCI_CLOSE
- MCI_INFO
- MCI_STATUS
- MCI_GETDEVCAPS

MCDs should be able to handle these commands at any time for any device context. Once the first RESTORE is complete, other commands can be processed.

Note: For detailed descriptions of MCIDRV_RESTORE and MCIDRV_SAVE, refer to "Subsystem Messages" in the *OS/2 Multimedia Programming Reference* .

Wait and Notify Flags

MCI_WAIT specifies that the application should not send a notification message. It also specifies that the driver should not return to the caller until the entire command has been completed.

MCI_NOTIFY specifies that upon completion of the command a notify message is to be sent to the application using the mdmDriverNotify function. If the notify flag is specified, the MCD should perform error checking and minimal processing on the command before returning to the caller. Error checking should ensure that the command can begin the main processing.

If neither MCI_WAIT or MCI_NOTIFY is specified, the processing is the same as if MCI_NOTIFY were specified, except that notification is not provided when the command completes.

Error and parameter checking should minimize the possibility that an error occurs after control is returned to the calling thread, requiring a notify error message to be sent to the calling application. The list of error conditions that are checked before returning to the caller will vary from driver to driver.

The main processing of the command should be done using a separate thread or a mechanism that allows control to be returned to the calling thread. If the MCD is using the SSM subsystem, the event procedure should be used for handling notify commands.

If an error is returned on an mciDriverEntry call, the error should not be sent using mdmDriverNotify. After an error is returned on mciDriverEntry, the MCD has finished processing that command.

On notify commands, it is the responsibility of the MCD to ensure that the data structure associated with the command message is copied into driver memory before returning to the caller. If this is not done, the driver might find that the memory was changed before it could process the command. For example, the *hwndCallback* , *ulFrom* , and *ulTo* fields on the MCI_PLAY message should be copied when the MCI_NOTIFY flag is specified.

Media control messages that can require a lot of time to complete are MCI_PLAY, MCI_RECORD, MCI_SEEK, and MCI_STEP. These commands are action commands. Messages such as MCI_PAUSE, MCI_GETDEVCAPS, MCI_STATUS, and MCI_SET are non-action commands and do not require much processing time. The non-action commands need not be processed on a separate thread if the notify flag is specified. For these commands, the call to mdmDriverNotify should be made at the completion of the command.

If the MCD is using the Stream Programming Interface (SPI) functions to perform data streaming, the SPI events should be used instead of separate threads. SPI requires an event routine to handle SPI events such as End-of-Stream, an error, and so on. SPI will call this event routine on one of its threads. Therefore, the mdmDriverNotify call could be made as part of the event routine.

An MM_MCINOTIFY message can have the notification codes shown in the following table associated with it.

Notification Code	Condition That Caused the Notification
MCI_NOTIFY_SUCCESSFUL	The command completed successfully with no errors.
MCI_NOTIFY_SUPERSEDED	A second command of the same type is received

	with the notify flag set.
MCI_NOTIFY_ABORTED	A second command is received that prevents the first command from completing successfully. If the driver finds an error early in the parameter-checking or command processing that requires an error to be returned to the caller, no notification is sent back to the application.

If the notification code field of MM_MCINOTIFY contains a value other than those shown in the previous table, the value is the error code for a specific OS/2 multimedia error condition. Only one MM_MCINOTIFY message per command should be sent.

Basic Command Messages

Basic commands are commands all device types should understand but can modify the parameters. For example, when issuing a PLAY command to a Videodisc Player, it might be desirable to indicate the speed of playback in frames per second. However, a device such as a CD Player might not have the capability to play back at different speeds.

The list of basic commands is shown in the following table. If a device does not use a device-type command, it can return MCIERR_UNSUPPORTED_FUNCTION. If a device supports the command, but not all of the options, it can return the MCIERR_UNSUPPORTED_FLAG for options that are not applicable.

Message	Description
MCI_CONNECTOR	Enables, disables, counts the number of, or queries the status of connectors.
MCI_LOAD	Loads a new device element (file name) into an open device context.
MCI_MASTERAUDIO	Sets the system master audio setting for all audio devices in the system. MCI_MASTERAUDIO is also used as a system command to query the current audio settings when the driver is first opened.
MCI_PAUSE	Suspends device playback.
MCI_PLAY	Starts playing the device.
MCI_RECORD	Starts recording data.
MCI_RESUME	Resumes playing or recording from a paused state.
MCI_SAVE	Saves data for the device.
MCI_SEEK	Moves to the specified position and stops.
MCI_SET	Sets the operating state of the device.
MCI_SETCUEPOINT	Sets a cuepoint.
MCI_SETPOSITIONADVISE	Sets a position change notification for the device.
MCI_STATUS	Obtains information about the status of a media device.
MCI_STOP	Stops the device.

The basic command messages use the *ulParam1* parameter for the flags applicable to the command message. They also use the *pParam2* parameter for a pointer to a message-specific data structure. Your MCD might add flags and parameters to create extended commands. When you extend a command message, your MCD must still respond to the basic flags and parameters.

The following table identifies the flags and data structures of the basic command messages. For a complete reference of media control interface commands, refer to the *OS/2 Multimedia Programming Reference* .

Message	Parameters (ulParam1)	Data Structure (pParam2)
MCI_CONNECTOR	MCI_NOTIFY MCI_WAIT MCI_ENABLE_CONNECTOR MCI_DISABLE_CONNECTOR MCI_QUERY_CONNECTOR_STATUS MCI_CONNECTOR_TYPE MCI_CONNECTOR_INDEX	MCI_CONNECTOR_PARMS
MCI_LOAD	MCI_OPEN_ELEMENT MCI_OPEN_MMIO MCI_NOTIFY MCI_WAIT	MCI_LOAD_PARMS
MCI_MASTERAUDIO	MCI_WAIT MCI_QUERYCURRENTSETTING MCI_QUERYSAVEDSETTING MCI_SAVESETTING MCI_MASTERVOL MCI_SPEAKERS MCI_HEADPHONES MCI_ON MCI_OFF	MCI_MASTERAUDIO_PARMS
MCI_PAUSE	MCI_NOTIFY MCI_WAIT	MCI_GENERIC_PARMS
MCI_PLAY	MCI_NOTIFY MCI_WAIT MCI_FROM MCI_TO	MCI_PLAY_PARMS
MCI_RECORD	MCI_NOTIFY MCI_WAIT MCI_FROM MCI_TO MCI_RECORD_INSERT MCI_RECORD_OVERWRITE	MCI_RECORD_PARMS
MCI_RESUME	MCI_NOTIFY MCI_WAIT	MCI_GENERIC_PARMS
MCI_SAVE	MCI_NOTIFY MCI_WAIT MCI_SAVE_FILE	MCI_SAVE_PARMS
MCI_SEEK	MCI_NOTIFY MCI_WAIT MCI_TO MCI_TO_START MCI_TO_END	MCI_SEEK_PARMS
MCI_SET	MCI_NOTIFY MCI_WAIT MCI_SET_AUDIO MCI_SET_DOOR_OPEN MCI_SET_DOOR_CLOSED MCI_SET_DOOR_LOCK MCI_SET_DOOR_UNLOCK MCI_SET_VOLUME MCI_OVER MCI_SET_VIDEO MCI_SET_ON MCI_SET_OFF MCI_SET_SPEED_FORMAT MCI_SET_TIME_FORMAT MCI_SET_ITEM	MCI_SET_PARMS
MCI_SETCUEPOINT	MCI_NOTIFY MCI_WAIT MCI_SET_CUEPOINT_ON MCI_SET_CUEPOINT_OFF	MCI_CUEPOINT_PARMS

MCI_SETPOSITIONADVISE	MCI_NOTIFY MCI_WAIT MCI_SET_POSITION_ADVISE_ON MCI_SET_POSITION_ADVISE_OFF	MCI_POSITION_PARMS
MCI_STATUS	MCI_NOTIFY MCI_WAIT MCI_STATUS_START MCI_TRACK MCI_STATUS_ITEM	MCI_STATUS_PARMS
MCI_STOP	MCI_NOTIFY MCI_WAIT	MCI_GENERIC_PARMS

System Command Messages

System commands are interpreted directly by the Media Device Manager (MDM), and are not passed to the MCDs. The following table lists the system commands that are supported by MDM.

Command	Description
MCI_ACQUIREDEVICE	Acquires use of the physical resources for the device.
MCI_CONNECTION	Returns information about the device context connections.
MCI_CONNECTORINFO	Returns information about the connectors on a device.
MCI_DEFAULTCONNECTION	Makes, breaks, or queries a default connection.
MCI_GROUP	Provides control over multiple devices with a single command.
MCI_MASTERAUDIO	Queries the current audio settings when the driver is first opened. This command is also used as a basic command to later adjust the master audio.
MCI_SYSINFO	Obtains information about devices installed in the system.
MCI_RELEASEDEVICE	Releases exclusive use of the physical resources for the devices.

Command Processing

All MCDs have commands passed to them in the same way, regardless of whether they support streaming or nonstreaming devices. Applications call either the mciSendCommand or mciSendString functions to pass commands to the MCD entry point, mciDriverEntry. In addition, some commands are generated by the MDM itself. These are the commands for saving and restoring an instance and synchronization messages-messages periodically generated by the master device for routing to the slave devices in the group.

Although streaming and nonstreaming devices have commands passed to them in the same way, they process the commands they receive from the MDM differently. These differences are described in the sections that follow and are summarized in the following table.

When the MDM sends:	The waveform audio MCD:	The CD audio MCD:
MCI_SET messages that set audio attributes	Routes the messages to the amp mixer.	Uses IOCtls to process the messages.
Messages that control playback (for example, SEEK, PLAY, PAUSE, STOP)	Routes the messages to the SSM.	Uses IOCtls to process the messages.

Error Return Codes

For MCDs to look consistent to applications, error codes should be consistent across MCDs. Following are some guidelines for using common error codes.

Use this error message:	When this error condition occurs:
MCIERR_OUTOFRANGE	The value given is out of range.
MCIERR_UNRECOGNIZED_COMMAND	Unknown *usMessage* value.
MCIERR_INVALID_FLAG	Unknown *ulParam1* value.
MCIERR_INVALID_ITEM_FLAG	Unknown flag in *ulItem* field.
MCIERR_INVALID_TIME_FORMAT_FLAG	Unknown flag in *ulTimeFormat* field.
MCIERR_SPEED_FORMAT_FLAG	Unknown flag in *ulSpeedFormat* field.
MCIERR_MISSING_PARAMETER	Invalid or NULL *pParam2* field.
MCIERR_INVALID_BUFFER	Invalid address to output buffer in *pParam2*.
MCIERR_MISSING_FLAG	Missing flags in *ulParam1* when one or more flags are required.
MCIERR_UNSUPPORTED_FLAG	Flag in *ulParam1* is valid for the message, but driver cannot perform the task.
MCIERR_UNSUPPORTED_FUNCTION	Function in *usMessage* is understood by the device but is not supported by the driver.
MCIERR_FLAGS_NOT_COMPATIBLE	More than one mutually exclusive flag is set.

The following error conditions are always handled by MDM:

- An invalid callback handle is specified for a message with the MCI_NOTIFY flag set.
- MCI_WAIT and MCI_NOTIFY flags are used together.
- A command is sent to an inactive instance.

Return Values and Return Types

MCDs should attempt to use the MCIERR return codes wherever possible. These return codes are provided in the C header file MEERROR.H. Error codes from SPI, MMIO or the operating system should be mapped into MCIERR return codes when it makes sense. If the driver has unique return codes for certain conditions, it can use error numbers MCIERR_CUSTOM_DRIVER_BASE through MEBASE-1. If custom driver return codes are used, a custom driver error table should be created.

MCDs should use the following return types in the high-order word of the return code when the media message has a return field.

Return Type	**Meaning**
MCI_INTEGER_RETURNED	Convert the given binary integer to its unsigned ASCII string representation.
MCI_COLONIZED2_RETURN	Convert the binary integer to the following form: `byte0:byte2`
MCI_COLONIZED3_RETURN	Convert the binary integer to the following form: `byte0:byte1:byte2`
MCI_COLONIZED4_RETURN	Convert the binary integer to the following form: `byte0:byte1:byte2:byte4`
MCI_TRUE_FALSE_RETURN	Convert the TRUE/FALSE value to the string representation.
MCI_ON_OFF_RETURN	Convert the ON/OFF value to the string representation.
MCI_DEVICENAME_RETURN	Convert the device type integer to its device name string.
MCI_TIME_FORMAT_RETURN	Convert the time format integer to its appropriate string.
MCI_SPEED_FORMAT_RETURN	Convert the speed format integer to its appropriate string.
MCI_MEDIA_TYPE_RETURN	Convert the disc type integer for a videodisc to "CLV," "CAV," "OTHER".
MCI_TRACK_TYPE_RETURN	Convert the track type integer to "AUDIO," "DATA," "OTHER".
MCI_CONNECTOR_TYPE_RETURN	Convert the connector type integer to its appropriate string.
MCI_CDXA_CHANNEL_ DESTINATION_RETURN	Convert the CDXA channel

	destination type integer to its appropriate string.
MCI_PREROLL_TYPE_RETURN	Convert the preroll type integer to its appropriate string.
MCI_FORMAT_TAG_RETURN	Convert the format tag type integer to its appropriate string.
MCI_SIGNED_INTEGER_RETURN	Convert the given binary integer to its signed ASCII string representation.

The return types described in the previous list allow the string parser to convert binary return values into strings. This conversion takes place only when the application calls mciSendString and only if no error occurs on the call. For example: "capability cdaudio has video" returns FALSE in the return string field of mciSendString. The media driver for the CD audio MCD puts a zero in the *ulReturn* field of the MCI_GETDEVCAPS_PARMS structure and MCI_TRUE_FALSE_RETURN in the high-order word of the return code.

Adding New Command Messages

Following are four steps a media driver author needs to follow to add support for a device-specific message to a driver:

1. Define the new or modified media control message your driver plans to support. (See Defining New Messages.)
2. Define the new data structure and flags for this message. (See Defining New Data Structures and Flags.)
3. Create a custom command table that MDM can use to parse the command string in mciSendString, so it can create the appropriate data structure to pass to your driver. (See Parsing a Command List).
4. Support the new message in your MCD's entry point (mciDriverEntry).

Defining New Messages

Suppose you want to define a new reset command for an MCD supporting a videodisc player. You can call this command by sending a **reset *program_number*** string. The command sets all player parameters to their default settings or one of several predefined states indicated by the *program_number* .

You would add the following message to the list of messages you must support in mciDriverEntry:

```
#define MCI_VD_RESET MCI_USER_MESSAGES
```

MCI_USER_MESSAGES is the first integer you can use for custom messages. Since the resource compiler does not accept mathematical expressions in the RCDATA resource type, you must indicate the specific value you want to use.

Now that you have defined the message ID, you must define the data structure and the command table, and create the code to handle this message in mciDriverEntry.

Defining New Data Structures and Flags

Most new messages require additional parameters to specify their exact function. The media control interface messages use *pParam2* as a pointer to a data structure, and use *ulParam1* as a bit-field for the flags associated with the message. A flag will exist for each field in the data structure that accepts data from the calling application. The application uses the flag to set the bit-field of *ulParam1* to indicate a value

is assigned to a particular field. Flags also specify options without parameters. Thus these flags do not correspond to a field in the data structure.

Defining a New Data Structure

The fields of the data structure for a media control interface message are always the size of ULONGs. The number of fields in the structure depends on the particular message. The first field must be reserved for a handle to a window function used with the MCI_NOTIFY flag. The next fields in the data structure depend on the type of data returned for the message.

- If no data is returned, no return fields are reserved in the data structure. Any data fields for passing information to the MCD immediately follow the *hwndCallback* field. For example, the MCI_SAVE_PARMS structure shown in the following figure does not have a return field.

```
typedef struct _MCI_SAVE_PARMS {
    HWND    hwndCallback;    /* PM window handle for MCI notify message */
    PSZ     pszFileName;     /* File name to save data to               */
 } MCI_SAVE_PARMS;
typedef MCI_SAVE_PARMS   *PMCI_SAVE_PARMS;
```

- If integer data is returned, the second field of the data structure is reserved for the return data. Any data fields for passing information to the MCD start in the third field. For example, the MCI_GETDEVCAPS_PARMS structure shown in the following example uses *ulReturn* as the integer return field.

```
typedef struct _MCI_GETDEVCAPS_PARMS {
    HWND    hwndCallback;    /* PM window handle for MCI notify message */
    ULONG   ulReturn;        /* Return field                            */
    ULONG   ulItem;          /* Item field for GETDEVCAPS item to query */
    USHORT  usMessage;       /* Field to hold MCI message to query      */
    USHORT  usReserved0;     /* Reserved field                          */
} MCI_GETDEVCAPS_PARMS;
typedef MCI_GETDEVCAPS_PARMS   * PMCI_GETDEVCAPS_PARMS;
```

- If string data is returned, the second and third fields of the data structure are reserved for the return data. The second ULONG is assigned to a field reserved for a pointer to the null-terminated return string. The third ULONG is assigned to a field reserved for the size of return buffer. The application is responsible for creating the buffer for return string. Any data fields for passing information to the MCD start in the fourth field. For example, the MCI_INFO_PARMS structure shown in the following example uses *pszReturn* and *ulRetSize* for the return fields.

```
typedef struct _MCI_INFO_PARMS {
    HWND      hwndCallback;  /* PM window handle for MCI notify message */
    PSZ       pszReturn;     /* Pointer to return buffer                */
    ULONG     ulRetSize;     /* Return buffer size                      */
} MCI_INFO_PARMS;
typedef MCI_INFO_PARMS   *PMCI_INFO_PARMS;
```

- If RECTL data is returned, the second through fifth fields of the data structure are reserved for the return data. The first ULONG position is reserved for the left values of the RECTL data. The second ULONG position is reserved for the bottom values of the RECTL data. Any data fields for passing information to the device driver start in the sixth ULONG position. Rather than specifying two ULONGs for the RECTL data, most data structure definitions use one RECTL data field to obtain an equivalent data structure. For example, the MCI_VID_RECT_PARMS structure shown in the following example uses *rc* for the return field.

```
typedef  struct _MCI_VID_RECT_PARMS {
    HWND    hwndCallback;    /* PM window handle for MCI notify message */
    RECTL   rc;              /* rectangle array specifying the offset   */
                             /* and size of a rectangle                 */
} MCI_VID_RECT_PARMS;
```

Assigning Flag Values

In addition to indicating the fields used in a data structure, flags can indicate an option that does not use any parameters. For example, the MCI_WAIT flag does not use any parameters.

When you add new flags, you must make sure that they do not conflict with the flags already in use. Bits 0 through 15 of the 32-bit ULONG are reserved for MDM. Bit 16 (0x00010000) is the first fit that a driver can use for its flags. If your command message is a new command message, you must choose bit positions that do not conflict with the flags already defined for that command. Any unused bits in the new flag must be set to zero. For example, if a new command for videodisc players uses flags 16 through 20, a custom extension to the new command could use flags 21 through 31.

To continue the example for adding a RESET command to a videodisc driver, the code sample shown in the following example defines a data structure, a corresponding pointer for it, and the flag corresponding to the program field of the data structure.

```
typedef struct {
  HWND   hwndCallback;
  ULONG  ulProgram;
} MCI_VD_RESET_PARMS;

typedef MCI_VD_RESET_PARMS FAR * PMCI_VD_RESET_PARMS;

#define MCI_VD_RESET_PROGRAM     0x00010000L
```

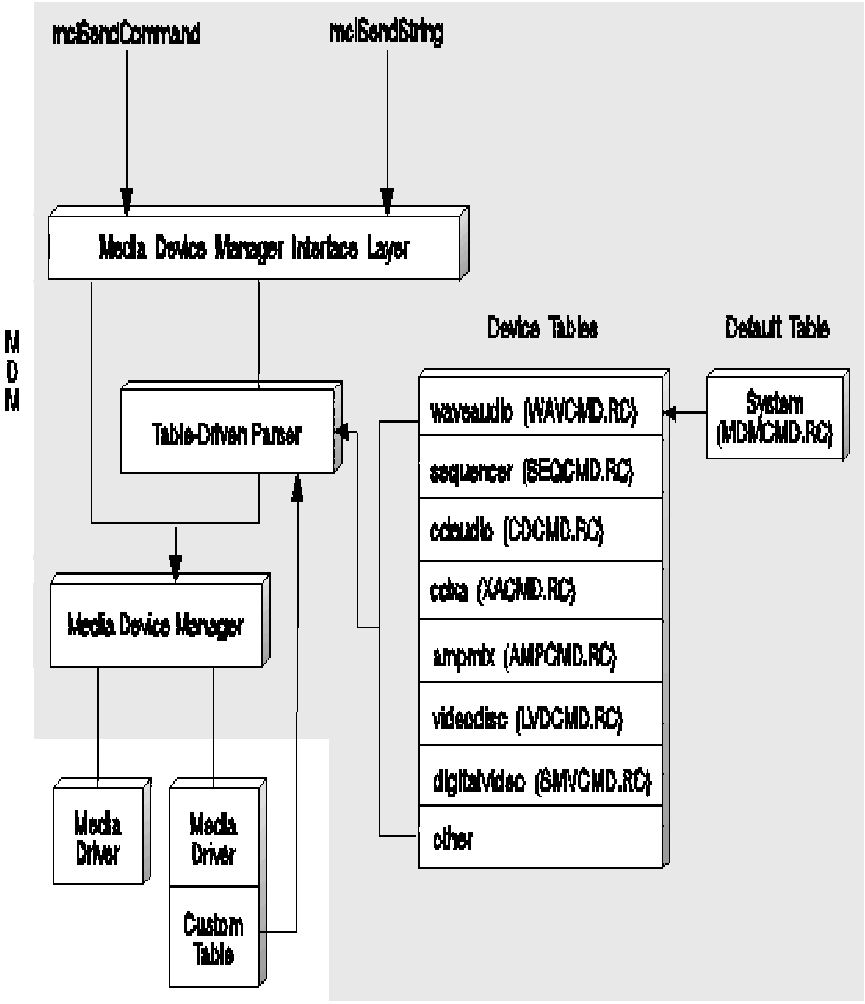
When the application sets the MCI_VD_RESET_PROGRAM flag in *ulParam1* , it indicates that a value is assigned in the *ulProgram* field.

Now that you have created the message command, data structure, and flags, you need to create the command table that tells MDM how to translate the equivalent string command into the message command format. See Parsing a Command List.

Command Tables

Command tables are structures that allow the MDM string parser to interpret command strings for MCDs. This provides string command interface support for your MCD. Represented as resources to the driver, command tables are created using the RCDATA type of resource. The resource number of the RCDATA block is the device type number.

The following figure illustrates how the Media Device Manager (MDM) uses commands tables. A program or mciRxInit command uses the mciSendString function to pass a string command to the MDM. In turn, MDM calls its string parser to interpret the string command and change it into the equivalent procedural command. MDM scans the command tables from the most specific to the least specific. It first scans any custom device-specific tables (if present in the MCD). Next, MDM looks for the string in the device-type table and finally the system default table.



MDM provides device-type tables to support OS/2 multimedia logical devices as shown in the above figure. These built-in tables will meet your MCD requirements unless you have to add or modify commands. If your MCD has new requirements (such as a new flag or a different data structure for a command), you must create a custom table to define commands specific to your device.

A custom table allows you to create your own versions of basic media control interface commands by adding new or changed command entries. For example, the device-type table for digital video supports the SEEK flags shown in the following example.

```
"seek",                 MCI_SEEK, 0,            MCI_COMMAND_HEAD,
   "notify",            MCI_NOTIFY,             MCI_FLAG,
   "wait",              MCI_WAIT,               MCI_FLAG,
   "to start",          MCI_TO_START,           MCI_FLAG,
   "to end",            MCI_TO_END,             MCI_FLAG,
   "to",                MCI_TO,                 MCI_INTEGER,
"",                     0L,                     MCI_END_COMMAND,
```

You, however, want your MCD to seek to the nearest position. To accomplish this, you can create a custom table in your MCD to parse commands for the digital video player (as shown in the following example. Note that the custom table must include *all* parameters associated with the changed command (not just new and changed parameters).

```
"seek",                 MCI_SEEK, 0,            MCI_COMMAND_HEAD,
   "notify",            MCI_NOTIFY,             MCI_FLAG,
   "wait",              MCI_WAIT,               MCI_FLAG,
   "to start",          MCI_TO_START,           MCI_FLAG,
   "to nearest",        MCI_TO_NEAREST,         MCI_FLAG,
   "to end",            MCI_TO_END,             MCI_FLAG,
   "to",                MCI_TO,                 MCI_INTEGER,
"",                     0L,                     MCI_END_COMMAND,
```

When writing an MCD, the DLL file names (which include the command table resources) must be referenced in the MMPM2.INI file. MDM

updates the INI file with MCD command table information you provide when installing your MCD. See Installing a Media Control Driver for details.

The following shows an example of how command table entries appear in the MMPM2.INI file. The MCDTABLE entry includes the MDM.DLL file, which includes the device-type command tables and the default system table. The VSDTABLE entry indicates that a custom command table resource is located in the SVMC.DLL file.

MDM scans the command tables from the most specific to the least specific. In the example below MDM first searches the custom table resources in the SVMC.DLL file. MDM next searches the internal digitalvideo device-type table (SMVCMD.RC) and then the system default table (MDMCMD.RC) located in the MDM.DLL file.

```
[ibmdigvid01]
  VERSIONNUMBER=1
  PRODUCTINFO=SOFTWARE MOTION VIDEO
  MCDDRIVER=SVMC
  VSDDRIVER=AUDIOIF
  PDDNAME=AUDIO1$
  MCDTABLE=MDM
  VSDTABLE=SVMC
  RESOURCENAME=DIGITALVIDEO
  DEVICEFLAG=1
  DEVICETYPE=12
  SHARETYPE=3
  RESOURCEUNITS=1
  RESOURCECLASSES=1,1
  VALIDCOMBINATIONS=
  EXTNAMES=3,UMB,MMM,V
  ALIASNAME=DV
```

Command Table Syntax

A command table consists of command lists. Each command list in the table defines the parsing of a particular command. For example, the following example shows the command lists for MCI_OPEN and MCI_INFO, which are part of the system default table contained in the MDMCMD.RC file.

```
"open",                 MCI_OPEN, 0,              MCI_COMMAND_HEAD,
   "",                  MCI_INTEGER,              MCI_RETURN,
   "notify",            MCI_NOTIFY,               MCI_FLAG,
   "wait",              MCI_WAIT,                 MCI_FLAG,
   "readonly",          MCI_READONLY,             MCI_FLAG,
   "shareable",         MCI_OPEN_SHAREABLE,       MCI_FLAG,
   "type",              0L,                       MCI_STRING,
   "",                  MCI_OPEN_ELEMENT,         MCI_STRING,
   "alias",             MCI_OPEN_ALIAS,           MCI_STRING,
"",                     0L,                       MCI_END_COMMAND,

"info",                 MCI_INFO, 0,              MCI_COMMAND_HEAD,
   "",                  MCI_STRING,               MCI_RETURN,
   "notify",            MCI_NOTIFY,               MCI_FLAG,
   "wait",              MCI_WAIT,                 MCI_FLAG,
   "product",           MCI_INFO_PRODUCT,         MCI_FLAG,
   "file",              MCI_INFO_FILE,            MCI_FLAG,
"",                     0L,                       MCI_END_COMMAND,
```

Command lists have a sequence of lines that are organized into three columns:

- Column 1 contains the command and its flags in command string format.
- Column 2 contains the command and its flags in command message format.
- Column 3 contains the Line_types for each line in the command list.

Line Types

The MDM string parser uses the Line_type to determine how to interpret each line. A command list begins with the MCI_COMMAND_HEAD Line_type and ends with the MCI_END_COMMAND Line_type.

MCI_RETURN

This Line_type indicates that the command provides return information. If used, MCI_RETURN must be the second entry in the command list.

- Column 1 contains **""**.
- Column 2 contains the type of return: integer or string.
- Column 3 contains MCI_RETURN.

A command returning an integer requires a ULONG for that return value. A command returning a string requires two ULONGs: the first ULONG is for a pointer to the return buffer, and the second ULONG is for the return buffer size.

The parser sends back return information in the *pszReturnString* and *usReturnLength* parameters of the mciSendString function. The above example shows how both types of returns are used.

The MCI_OPEN returns a ULONG value, and MCI_INFO returns data that is filled into the buffer pointed to by the pointer in the second ULONG of the command structure. (Remember that the first ULONG is always *hwndCallback*.)

MCI_RETURN_TYPE

This Line_type is used to convert return values into strings by the string parser.

```
"",             MCI_TRUE_FALSE_RETURN, 0,    MCI_RETURN_TYPE,
   "TRUE",      1L,                          MCI_RETURN_TYPE_STRING,
   "FALSE",     0L,                          MCI_RETURN_TYPE_STRING,
"",             0L,                          MCI_END_RETURN_TYPE,
```

The *flag_define* field for the return line is compared against the high-order word of the return code. If a match is found, then the parser tries to find a match for the *flag_define* field and the return value. If a match is found for the return value, then the keyword string is returned to the application in the *pszReturnString* field of the mciSendString call.

MCI_FLAG

This Line_type is used to convert keywords to flag values for *ulParam1*. When more than one MCI_FLAG Line_type is encountered during the parsing of a command string, the *ulParam1* parameter is ORed with the *flag_define* field of the *Keyword_list*.

MCI_STRING

This Line_type is used for string values for a message. The string following the MCI_STRING keyword is copied into the command structure. MDM allocates memory to hold the string and put the address into the command structure.

MCI_INTEGER

This Line_type is used for integer values for a message. The integer following the MCI_INTEGER keyword is copied into the command structure. The integer value is converted from a string representation to a ULONG.

MCI_CONSTANT

This Line_type and the MCI_END_CONSTANT Line_type allow a set of possible keywords to represent a flag or integer. In the following example, any one of the time format values can be used as the value for the time format constant.

```
"",                          MCI_SET_TIME_FORMAT,     MCI_CONSTANT,
   "time format milliseconds", MCI_FORMAT_MILLISECONDS, MCI_INTEGER,
   "time format ms",           MCI_FORMAT_MILLISECONDS, MCI_INTEGER,
   "time format mmtime",       MCI_FORMAT_MMTIME,       MCI_INTEGER,
"",                          0L,                      MCI_END_CONSTANT,
```

The value on the MCI_CONSTANT line_type is ORed with the *ulParam1* field. The values on the MCI_INTEGER line_types within this constant block are copied into the command structure. The constant block represents just one ULONG in the command structure.

MCI_DEFAULT_STRING

This Line_type and the MCI_DEFAULT_INTEGER Line_type provide the means for an unknown value to be used as a string or integer. For example the LOAD command (shown in the following example) uses a default string for the filename string.

```
"load",                          MCI_LOAD, 0,           MCI_COMMAND_HEAD,
   "notify",                     MCI_NOTIFY,            MCI_FLAG,
   "wait",                       MCI_WAIT,              MCI_FLAG,
   "new",                        0L,                    MCI_FLAG,
   "readonly",                   MCI_READONLY,          MCI_FLAG,
   "",                           MCI_OPEN_ELEMENT,      MCI_DEFAULT_STRING,
"",                              0L,                    MCI_END_COMMAND,
```

MCI_RECTL

This Line_type specifies that RECTL data modifies the media control interface flag represented by the command list entry. The second column of this entry contains the flag to set. The data structure for the message command must reserve a ULONG to hold the integer value.

MCI_OR

This Line_type allows you to combine different line types within the same construct. The following shows an example of an MCI_OR entry used in the MCI_STATUS command list in the MDMCMD.RC file. Notice that the *%d* symbol can be used in a command list as an indicator to print the decimal value.

```
   "",              0L,                          MCI_OR,
   "track",         MCI_TRACK,                   MCI_INTEGER,
   "channel",       0L,                          MCI_CONSTANT,
      "all",        MCI_STATUS_AUDIO_ALL,        MCI_INTEGER,
      "left",       MCI_STATUS_AUDIO_LEFT,       MCI_INTEGER,
      "right",      MCI_STATUS_AUDIO_RIGHT,      MCI_INTEGER,
      "%d",         0L,                          MCI_INTEGER,
   "",              0L,                          MCI_END_CONSTANT,
   "",              0L,                          MCI_END_OR,
```

MCI_STRING_LIST

Pointer to an array of pointers to strings in the list. For example, the following example code uses MCI_STRING_LIST to point to an array of pointers to device identifiers.

```
"group",            MCI_GROUP, 0,                MCI_COMMAND_HEAD,
   "delete",        MCI_GROUP_DELETE,            MCI_FLAG,
   "nopiecemeal",   MCI_NOPIECEMEAL,             MCI_FLAG,
   "synchronize",   MCI_SYNCHRONIZE,             MCI_FLAG,
   "wait",          MCI_WAIT,                    MCI_FLAG,
   "notify",        MCI_NOTIFY,                  MCI_FLAG,
   "",              0L,                          MCI_INTEGER,
   "",              0L,                          MCI_INTEGER,
   "master",        MCI_GROUP_MASTER,            MCI_STRING,
   "",              0L,                          MCI_STRING,
   "",              0L,                          MCI_INTEGER,
   "make",          MCI_GROUP_MAKE,              MCI_STRING_LIST,
"",                 0L,                          MCI_END_COMMAND,
```

Parsing a Command List

A command table consists of command lists, which define to the MDM parser how to parse certain commands. For example, illustrates a command list for MCI_SEEK:

```
"seek",                 MCI_SEEK, 0,          MCI_COMMAND_HEAD,
   "notify",            MCI_NOTIFY,           MCI_FLAG,
   "wait",              MCI_WAIT,             MCI_FLAG,
   "to start",          MCI_TO_START,         MCI_FLAG,
   "to end",            MCI_TO_END,           MCI_FLAG,
   "to",                MCI_TO,               MCI_INTEGER,
"",                     0L,                   MCI_END_COMMAND,
```

The SEEK command list tells the parser how to create the associated data structure for the *pParam2* parameter of mciSendCommand to point to, when it finds "seek" in the *pszCommand* parameter of mciSendString.

Notice that each line is broken into a null-terminated string and two ULONGs. The ULONG following "seek" is composed of two parts; MCI_SEEK and 0. The command message (in this case, MCI_SEEK) is a USHORT, and therefore we need another USHORT as a filler. The MCI_COMMAND_HEAD Line_type tells MDM that the first ULONG (actually, the first USHORT) contains the *usMessage* parameter for mciDriverEntry.

If the line type of a line is MCI_FLAG, then the first ULONG for that line is ORed together with all the other ULONGs for MCI_FLAG lines, to form the *ulParam1* parameter.

The string parser makes two passes through the command list. The first pass is to determine the size of the structure to allocate for the command. The second pass parses the command string, fills in the command structure, and creates a *ulParam1* flag.

Each command is composed of a number of ULONGs. All commands have a minimum of one ULONG for the *hwndCallback* field. The parser provides this ULONG automatically. The remaining size of the command structure is based on the Line_types used for the command.

If the line type of a line is MCI_INTEGER, then the text following string S, where S is the null-terminated string in the command line, in *pszCommand* is an integer and should be put in the corresponding ULONG in the structure of ULONGs that *pParam2* points to.

If the line type is MCI_STRING, then the text following string S in *pszCommand* is a string, and a pointer to this string will be put in the corresponding ULONG in the structure of ULONGs that *pParam2* points to.

Finally, if the line type is MCI_RETURN, then the first ULONG of that line specifies the value that is to be returned through the data structure pointed to by *pParam2* .

The *pParam2* data structure is laid out based on the command list. The first ULONG is always *hwndCallback* . The next one or two ULONGs represent MCI_RETURN:

- If MCI_RETURN is specified, and MCI_INTEGER is the first ULONG of that line, then the second ULONG of *pParam2* is *ulReturn* .
- If MCI_RETURN is specified and MCI_STRING is the first ULONG of that line, then the second ULONG of *pParam2* is *ulReturn* , and the third ULONG is *ulRetSize* . The *ulRetSize* specifies the length of the string.

The other fields of *pParam2* are filled in based on the MCI_INTEGER and MCI_STRING appearing in the second ULONG of the command lines. In addition, MCI_CONSTANT and MCI_END_CONSTANT form a range of possible values for one of the fields in the structure. This block represents a single ULONG in the structure. These fields are filled in, in the order they appear in the command list. For example, suppose the string parser parses the MCI_STATUS command list shown in the following example code.

```
"status\0",            MCI_STATUS, 0,                MCI_COMMAND_HEAD,
"\0",                  MCI_INTEGER,                  MCI_RETURN,
"notify\0",            MCI_NOTIFY,                   MCI_FLAG,
"wait\0",              MCI_WAIT,                     MCI_FLAG,
"\0",                  MCI_STATUS_ITEM,              MCI_CONSTANT,
"mode\0",              MCI_STATUS_MODE,              MCI_INTEGER,
"ready\0",             MCI_STATUS_READY,             MCI_INTEGER,
"current track\0",     MCI_STATUS_CURRENT_TRACK,     MCI_INTEGER,
"length\0",            MCI_STATUS_LENGTH,            MCI_INTEGER,
"number of tracks\0",  MCI_STATUS_NUMBER_OF_TRACKS,  MCI_INTEGER,
"position\0",          MCI_STATUS_POSITION,          MCI_INTEGER,
"position in track\0", MCI_STATUS_POSITION_IN_TRACK, MCI_INTEGER,
"time format\0",       MCI_STATUS_TIME_FORMAT,       MCI_INTEGER,
"speed format\0",      MCI_STATUS_SPEED_FORMAT,      MCI_INTEGER,
"\0",                  0L,                           MCI_END_CONSTANT,
"track\0",             MCI_TRACK,                    MCI_INTEGER,
"\0",                  0L,                           MCI_END_COMMAND,
```

The string parser produces the *pParam2* parameter shown in the following figure.

```
{
HWND       hwndCallback;
ULONG      ulReturn;
ULONG      ulItem;
ULONG      ulTrack;
}
```

Notice that the MCI_CONSTANT block is defined in order to fill in the *ulItem* field of the status structure. If the keyword field on the MCI_CONSTANT line is "\0" or NULL, then the parser is looking for any keyword defined in the block, and the first ULONG of the matching keyword is put in the *ulItem* field. If the keyword on the MCI_CONSTANT line is not "\0" then the parser looks for that keyword and ORs the first ULONG of the MCI_CONSTANT line with *ulParam1* . The parser continues to look for a keyword match in the constant block. For example:

```
"capability",              MCI_GETDEVCAPS, 0,                    MCI_COMMAND_HEAD,
 "",                       MCI_PREROLL_TYPE_RETURN, 0,           MCI_RETURN_TYPE,
    "deterministic",       MCI_PREROLL_DETERMINISTIC,            MCI_RETURN_TYPE_STRING,
    "notified"             MCI_PREROLL_NOTIFIED,                 MCI_RETURN_TYPE_STRING,
    "none",                MCI_PREROLL_NONE,                     MCI_RETURN_TYPE_STRING,
 "none",                   0L,                                   MCI_END_RETURN_TYPE,
 "",                       MCI_TRUE_FALSE_RETURN, 0,             MCI_RETURN_TYPE,
    "TRUE",                1L,                                   MCI_RETURN_TYPE_STRING,
    "FALSE",               0L,                                   MCI_RETURN_TYPE_STRING,
 "",                       0L,                                   MCI_END_RETURN_TYPE,
 "",                       MCI_DEVICENAME_RETURN, 0,             MCI_RETURN_TYPE,
    "Videotape",           MCI_DEVTYPE_VIDEOTAPE, 0,             MCI_RETURN_TYPE_STRING,
    "Videodisc",           MCI_DEVTYPE_VIDEODISC, 0,             MCI_RETURN_TYPE_STRING,
    "CDaudio",             MCI_DEVTYPE_CD_AUDIO,  0,             MCI_RETURN_TYPE_STRING,
    "DAT",                 MCI_DEVTYPE_DAT,       0,             MCI_RETURN_TYPE_STRING,
    "Audiotape",           MCI_DEVTYPE_AUDIO_TAPE,        0,    MCI_RETURN_TYPE_STRING,
    "Other",               MCI_DEVTYPE_OTHER,             0,    MCI_RETURN_TYPE_STRING,
    "Waveaudio",           MCI_DEVTYPE_WAVEFORM_AUDIO,    0,    MCI_RETURN_TYPE_STRING,
    "Sequencer",           MCI_DEVTYPE_SEQUENCER,         0,    MCI_RETURN_TYPE_STRING,
    "Ampmix",              MCI_DEVTYPE_AUDIO_AMPMIX,      0,    MCI_RETURN_TYPE_STRING,
    "Overlay",             MCI_DEVTYPE_OVERLAY,           0,    MCI_RETURN_TYPE_STRING,
    "Digitalvideo",        MCI_DEVTYPE_DIGITAL_VIDEO,     0,    MCI_RETURN_TYPE_STRING,
    "Speaker",             MCI_DEVTYPE_SPEAKER,           0,    MCI_RETURN_TYPE_STRING,
    "Headphone",           MCI_DEVTYPE_HEADPHONE,         0,    MCI_RETURN_TYPE_STRING,
    "Microphone",          MCI_DEVTYPE_MICROPHONE,        0,    MCI_RETURN_TYPE_STRING,
    "Monitor",             MCI_DEVTYPE_MONITOR,           0,    MCI_RETURN_TYPE_STRING,
    "CDXA",                MCI_DEVTYPE_CDXA,              0,    MCI_RETURN_TYPE_STRING,
 "",                       0L,                                   MCI_END_RETURN_TYPE,
 "",                       MCI_INTEGER,                          MCI_RETURN,
 "notify",                 MCI_NOTIFY,                           MCI_FLAG,
 "wait",                   MCI_WAIT,                             MCI_FLAG,
 "",                       MCI_GETDEVCAPS_ITEM,                  MCI_CONSTANT,
    "can record",          MCI_GETDEVCAPS_CAN_RECORD,            MCI_INTEGER,
    "can insert",          MCI_GETDEVCAPS_CAN_RECORD_INSERT,     MCI_INTEGER,
    "has audio",           MCI_GETDEVCAPS_HAS_AUDIO,             MCI_INTEGER,
    "has video",           MCI_GETDEVCAPS_HAS_VIDEO,             MCI_INTEGER,
    "can eject",           MCI_GETDEVCAPS_CAN_EJECT,             MCI_INTEGER,
    "can play",            MCI_GETDEVCAPS_CAN_PLAY,              MCI_INTEGER,
    "can save",            MCI_GETDEVCAPS_CAN_SAVE,              MCI_INTEGER,
    "uses files",          MCI_GETDEVCAPS_USES_FILES,            MCI_INTEGER,
    "compound device",     MCI_GETDEVCAPS_USES_FILES,            MCI_INTEGER,
    "can lockeject",       MCI_GETDEVCAPS_CAN_LOCKEJECT,         MCI_INTEGER,
    "can setvolume",       MCI_GETDEVCAPS_CAN_SETVOLUME,         MCI_INTEGER,
    "preroll type",        MCI_GETDEVCAPS_PREROLL_TYPE,          MCI_INTEGER,
    "preroll time",        MCI_GETDEVCAPS_PREROLL_TIME,          MCI_INTEGER,
    "device type",         MCI_GETDEVCAPS_DEVICE_TYPE,           MCI_INTEGER,
    "can stream",          MCI_GETDEVCAPS_CAN_STREAM,            MCI_INTEGER,
    "can process internal",MCI_GETDEVCAPS_CAN_PROCESS_INTERNAL,  MCI_INTEGER,
 "",                       0L,                                   MCI_END_CONSTANT,
 "message",                MCI_GETDEVCAPS_MESSAGE,               MCI_CONSTANT,
    "acquire",             MCI_ACQUIREDEVICE, 0,                 MCI_INTEGER,
    "release",             MCI_RELEASEDEVICE, 0,                 MCI_INTEGER,
    "open",                MCI_OPEN, 0,                          MCI_INTEGER,
    "close",               MCI_CLOSE, 0,                         MCI_INTEGER,
    "escape",              MCI_ESCAPE, 0,                        MCI_INTEGER,
    "play",                MCI_PLAY, 0,                          MCI_INTEGER,
    "seek",                MCI_SEEK, 0,                          MCI_INTEGER,
    "stop",                MCI_STOP, 0,                          MCI_INTEGER,
    "pause",               MCI_PAUSE, 0,                         MCI_INTEGER,
    "info",                MCI_INFO, 0,                          MCI_INTEGER,
    "capability",          MCI_GETDEVCAPS, 0,                    MCI_INTEGER,
    "status",              MCI_STATUS, 0,                        MCI_INTEGER,
    "spin",                MCI_SPIN, 0,                          MCI_INTEGER,
    "set",                 MCI_SET, 0,                           MCI_INTEGER,
    "step",                MCI_STEP, 0,                          MCI_INTEGER,
    "record",              MCI_RECORD, 0,                        MCI_INTEGER,
    "sysinfo",             MCI_SYSINFO, 0,                       MCI_INTEGER,
    "save",                MCI_SAVE, 0,                          MCI_INTEGER,
    "cue",                 MCI_CUE, 0,                           MCI_INTEGER,
    "update",              MCI_UPDATE, 0,                        MCI_INTEGER,
    "setcuepoint",         MCI_SET_CUEPOINT, 0,                  MCI_INTEGER,
    "setpositionadvise",   MCI_SET_POSITION_ADVISE, 0,           MCI_INTEGER,
    "setsyncoffset",       MCI_SET_SYNC_OFFSET, 0,               MCI_INTEGER,
    "load",                MCI_LOAD, 0,                          MCI_INTEGER,
    "masteraudio",         MCI_MASTERAUDIO, 0,                   MCI_INTEGER,
    "gettoc",              MCI_GETTOC, 0,                        MCI_INTEGER,
    "connector",           MCI_CONNECTOR, 0,                     MCI_INTEGER,
    "resume",              MCI_RESUME, 0,                        MCI_INTEGER,
```

```
 "",                    0L,                                MCI_END_CONSTANT,
"",                     0L,                                MCI_END_COMMAND,
```

The *pParam2* parameter would look like:

```
       {
       ULONG      hwndCallback;
       ULONG      ulReturn;
       ULONG      ulltem;
       ULONG      ulMessage;
       }
```

Notice the second constant block with the keyword "message". The parser would be looking for something like "message open". This constant block would OR the MCI_GETDEVCAPS_MESSAGE with the *ulParam1* parameter and put the value MCI_OPEN in the *ulMessage* field.

The multimedia string parser also supports one default INTEGER and STRING value. The constants MCI_DEFAULT_INTEGER and MCI_DEFAULT_STRING are used by the string parser to locate unknown keywords. For example, if a command needed a file name as its only parameter, then the following line could be used in the command table.

```
   "\0",            MCI_FILENAME,                MCI_DEFAULT_STRING,
```

This item takes up a ULONG in the structure, just as in MCI_STRING.

Support for device-specific command tables is provided by the multimedia installation application. When a device is installed, one of the parameters to the install process is the command table (resource DLL). The required and device-type command tables are provided by the system.

Error Tables

In addition to custom command tables, MCDs can have custom error tables. Error tables are represented as resources to the driver. They are created using the RCDATA type of resource. By default, custom driver error tables are associated with the MCDTABLE and VSDTABLE DLLs specified in the MMPM2.INI file. The resource number is the device type number + MMERROR_TABLE_BASE. For example, the following example shows an RCDATA value of 506. This consists of:

- A value of 500 from MMERROR_TABLE_BASE located in the MCIDRV.H file
- A value of 6 from the device type you are supporting (for example, MCI_DEVTYPE_OTHER located in the MCIOS2.H file).

As an alternate, you can add a #define statement in your header file to define the RCDATA value. For example:

```
#define MMERR_TABLE_MY_DEVICE  MMERROR_TABLE_BASE + MCI_DEVTYPE_OTHER
```

MDM attempts to use these tables whenever mciGetErrorString is called with an error generated by that driver. If the custom error table resource does not exist, then MDM uses its default error table. The error table is a resource.

Error tables are composed of a set of ULONG strings with a special end-of-table identifier as shown in the following example.

```
RCDATA        506
BEGIN

MCIERR_INVALID_DEVICE_NAME,        "Invalid Device Name given"
MCIERR_SUCCESS,                    "MMPM Command completed successfully"
MCIERR_INVALID_DEVICE_ID,          "Invalid device ID given"
MCIERR_UNRECOGNIZED_KEYWORD,       "Unrecognized keyword"
MCIERR_UNRECOGNIZED_COMMAND,       "Unrecognized command"
MCIERR_HARDWARE,                   "Hardware error"
MCIERR_OUT_OF_MEMORY,              "System out of memory"
     .
     .
     .
MCIERR_MSG_TABLE_END               ""
```

```
END
```

Note: If a device does not use a basic command, the MCD can return MCIERR_UNSUPPORTED_FUNCTION. If a device supports the command, but not all of the options, it can return MCIERR_UNSUPPORTED_FLAG.

Device States

Media devices that transport data are considered to be in one of the following nine states at any given time:

- Playing
- Recording
- Seeking
- Stopped
- Paused during a playback operation
- Paused during a recording operation
- Cued for a playback operation
- Cued for a recording operation
- Closed.

When a device is opened, the device context is assumed to be in the *stopped* state. The *closed* state can be viewed as both the initial state and the termination state. Or this state can be thought of as not a state at all, because a device context does not exist before it is opened, and ceases to exist when it is closed.

The following figure lists the allowable device states in the first column of the table and indicates the changes in state that occur when the command messages shown at the top of the table are issued. This table assumes all error conditions keep the device in its current state. For example, a waveform player that is opened without an element remains in the stopped state when a play is issued, and the MCD receives an error code.

Note that a device is in the *seek* state only during the SEEK operation. Upon completion of the seek, the device enters the *stop* state. The following figure is provided as a guide to application developers and MCD writers. There can be no guarantee that every media device will conform to this table, but every effort should be made to hide the complexity of the device from the application.

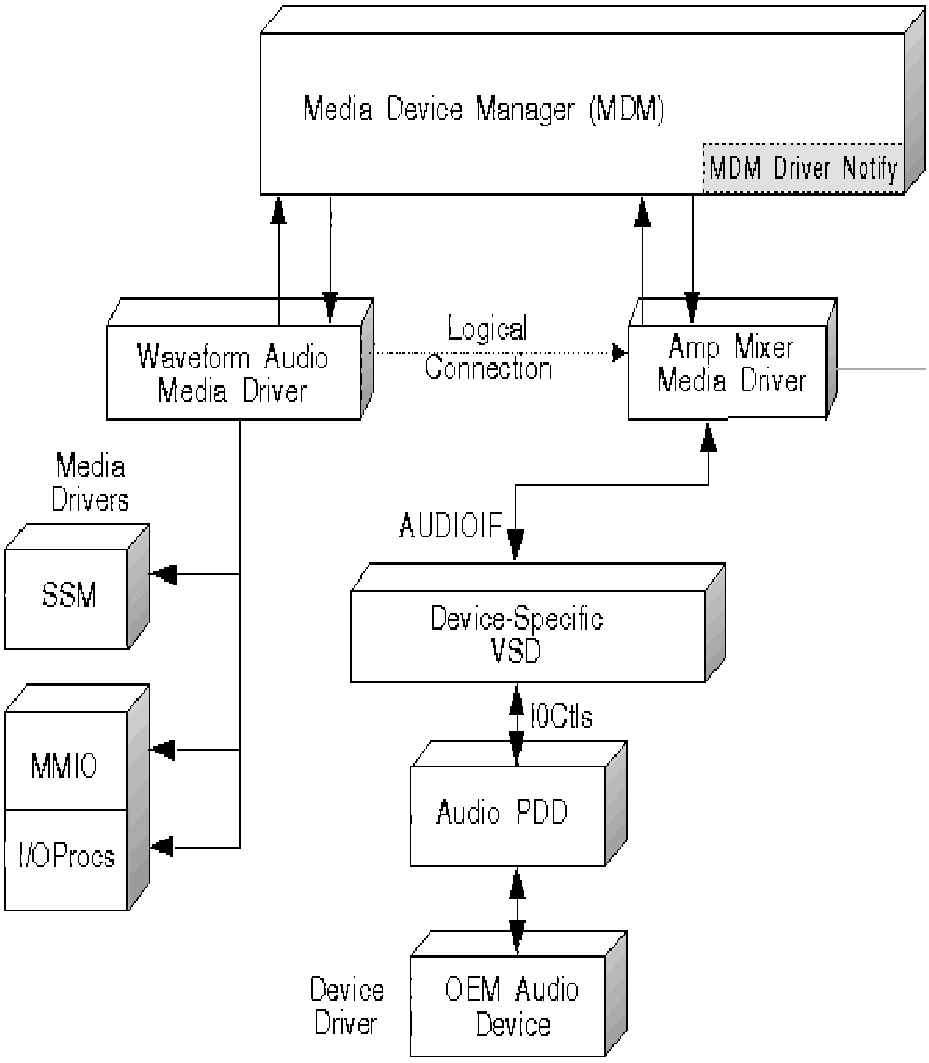
	MESSAGE / STATE	CLOSE	CONNECTOR	CUE|PLY	CUE|REC	LOAD	OPEN	PAUSE	PLAY	RECORD	RESUME	SEEK	SPIN|UP	SPIN|DN	STEP	
1	PLAY	0	E	7	8	4	-	5	1	2	1	3	1	4	5	
2	RECORD	0	E	7	8	4	-	6	1	2	2	3	-	4	6	
3	SEEK	0	E	7	8	4	-	3	1	2	3	3	7	4	E	
4	STOP	0	4	7	8	4	-	4	1	2	4	3	7	4	4	
5	PAUSE_PLAY	0	E	7	8	4	-	5	1	2	1	3	7	4	5	
6	PAUSE_REC	0	E	7	8	4	-	6	1	2	2	3	-	4	6	
7	CUE_PLAY	0	E	7	8	4	-	7	1	2	7	8	7	4	7	
8	CUE_REC	0	E	7	8	4	-	8	1	2	8	3	-	4	8	
9	CLOSE	-	-	-	-	-	4	-	-	-	-	-	-	-	-	

Legend

E Error condition.
- Not applicable.

Controlling a Streaming Device: Waveform Audio MCD

To play back a waveform file stored on a user's hard disk, the Duet Player I sample program sends command messages to the Media Device Manager (MDM) by means of the mciSendCommand function. The application is shielded from having to know specific information about the hardware. In other words, the application does not need to know which audio adapter is being used to play back the waveform file. The MDM sends the commands to a generic mciDriverEntry interface for audio devices, located in the AUDIOMCD DLL. The device-independent AUDIOMCD and its device-specific counterpart AUDIOIF are two of the modules that comprise the waveform audio MCD as shown in the following figure. Also depicted are the MMIO and SSM interfaces used by the MCD.



The Waveform Audio to Amp-Mixer Connection

Because its primary purpose is to play and record waveform data, the waveform audio MCD routes an MCI_SET application request to set audio attributes. The waveform audio MCD can route requests to the amp mixer because a default logical connection exists from the waveform audio MCD to the ampmixer MCD.

A default connection is the name of a connected device, while a device-context connection is the actual handle to a particular instance of an opened device. The waveaudio device has a default connection to an ampmixer device. When the waveaudio device is opened, it automatically opens the ampmix device creating an instance of each device. Since devices may be shared in OS/2 multimedia, the waveaudio device can be opened again by another application and two new instances are created. While the default connection is the same in both cases, the device context connections are different.

The code fragment in the following example illustrates how the waveaudio device gets a default connection and opens an associated amp mixer. The waveform audio MCD uses the device handle of the amp mixer to route any requests to change the volume.

An application can retrieve the amp mixer handle by issuing the MCI_CONNECTION message. This is necessary if the application needs to use any advanced audio shaping functions such as treble, bass, balance, and so on.

```
  ulpInstance->usWaveDeviceID = pDrvOpenParams->usDeviceID;

  ulDeviceTypeID = MAKEULONG ( MCI_DEVTYPE_WAVEFORM_AUDIO,
                               pDrvOpenParams->usDeviceOrd);
  /*************************************************
```

```
* Ensure that the INI file contains the right device id
**************************************************/

if ( pDrvOpenParams->usDeviceType != MCI_DEVTYPE_WAVEFORM_AUDIO )
   {
   return ( MCIERR_INI_FILE );
   }

usConnLength = sizeof(DEFAULTCONNECTIONS2);

ulrc =  mciQueryDefaultConnections ( ulDeviceTypeID,
                                     &DefCon,
                                     &usConnLength);
/**************************************************
* Ensure that the INI file says that we are connected
* to an amp mixer.  If it says that we are connected
* to ourselves, return an error.
**************************************************/
if ( ULONG_LOWD( DefCon.dwDeviceTypeID2 ) == MCI_DEVTYPE_WAVEFORM_AUDIO )
   {
   return ( MCIERR_INI_FILE );
   }
```

MMIO Operations

The waveform audio MCD uses MMIO functions to:

- Open the waveform data file
- Initialize the audio device with information obtained from the file's header
- Support multiple file formats
- Support networking functions
- Support editing functions.

Initializing the Audio Device

To initialize the audio device, the waveform audio MCD needs information about the data element to be played. The driver makes a call to mmioGetHeader, which returns information contained in the header of the waveform element (for example, whether the data was recorded in stereo or mono, what its sampling rate and bits-per-sample are, and so on.). AUDIOMCD parses this information and puts it into the instance structure. Next, the instance structure is passed to AUDIOIF, which uses the information to initialize the device by means of the IOCtl interface.

```
/**************************************************
* A streaming MCD should utilize MMIO to perform all
* file manipulations.  If we use MMIO, then the MCD
* will be free from file dependencies (that is, if a RIFF
* IOProc or a VOC IOProc is loaded will be irrelevant.
**************************************************/

ulrc = mmioGetHeader ( ulpInstance->hmmio,
                       (PVOID) &ulpInstance->mmAudioHeader ,
                       sizeof( ulpInstance->mmAudioHeader ),
                       (PLONG) &BytesRead,
                       (ULONG) NULL,
                       (ULONG) NULL);

if ( ulrc == MMIO_SUCCESS )
```

```
      {

      /****************************************
      * Copy the data from the call into the instance
      * so that we can set the amp/mixer up with the
      * values that the file specifies.
      ****************************************/

      AMPMIX.sMode            = WAVEHDR.usFormatTag;
      AMPMIX.sChannels        = WAVEHDR.usChannels;
      AMPMIX.lSRate           = WAVEHDR.ulSamplesPerSec;
      AMPMIX.lBitsPerSRate    = WAVEHDR.usBitsPerSample;
      ulpInstance->ulDataSize = XWAVHDR.ulAudioLengthInBytes;
      AMPMIX.ulBlockAlignment = ( ULONG )WAVEHDR.usBlockAlign;
      ulpInstance->ulAverageBytesPerSec = WAVEHDR.usChannels *
         WAVEHDR.ulSamplesPerSec * ( WAVEHDR.usBitsPerSample / 8 );

      } /* SuccesFul GetHeader */

    else
      {
      ulrc = mmioGetLastError( ulpInstance->hmmio );

      }
      return (ulrc);

}  /* GetAudioHeader */
```

The handle *hmmio* that is passed to mmioGetHeader was returned by the call to mmioOpen to open the data element. The buffer pointed to by *mmAudioHeader* is filled in by the MMIO Manager on return from mmioGetHeader. It is an MMAUDIODATA structure. When the MMAUDIODATA structure is returned by mmioGetHeader, it contains all the information the waveform audio MCD needs to initialize the audio device. The information includes:

- The type of waveform data in the file (*usFormatTag*)
- Whether the file was recorded in mono or stereo (*usChannels*)
- The sample rate and size set when the file was recorded (*usSamplesPerSec* and *usBitsPerSample*)
- The length of the audio data (*ulAudioLengthInBytes*).

Supporting Multiple File Formats

The following example shows how the waveform audio MCD searches audio I/O procedures by defining MMIO_MEDIATYPE_AUDIO in the *ulMedType* field of the MMCTOCENTRY data structure. To search and identify other types of IOProcs, you can include one of the following media type flags:

- MMIO_MEDIATYPE_IMAGE
- MMIO_MEDIATYPE_AUDIO
- MMIO_MEDIATYPE_MIDI
- MMIO_MEDIATYPE_COMPOUND
- MMIO_MEDIATYPE_OTHER
- MMIO_MEDIATYPE_UNKNOWN
- MMIO_MEDIATYPE_DIGITALVIDEO
- MMIO_MEDIATYPE_ANIMATION
- MMIO_MEDIATYPE_MOVIE

Refer to the *OS/2 Multimedia Programming Reference* for further information.

```
           mmioinfo.aulInfo[ 3 ] = MMIO_MEDIATYPE_AUDIO;

           /* Open the file */

           pInstance->hmmio = mmioOpen ( pFileName,
                                         &mmioinfo,
```

```
                                        ulFlags);

            /* Check for errors--see comments from above */

            if (pInstance->hmmio == (ULONG) NULL)
              {
              if ( mmioinfo.ulErrorRet == MMIOERR_MEDIA_NOT_FOUND )
                {
                return  ( MCIERR_INVALID_MEDIA_TYPE );
                }

              return  ( mmioinfo.ulErrorRet );
              }

            pInstance->ulCapabilities = 0;

      else
          {
          /*********************************************
          * Since the wave IOProc opened the file, we know
          * that it has the following capabilities.
          *********************************************/
          pInstance->ulCapabilities =  ( CAN_INSERT | CAN_DELETE | CAN_UNDOREDO +
                                         CAN_SAVE   | CAN_INSERT | CAN_RECORD  );
          }

    /***************************************
    * Get The Header Information
    ***************************************/

    if ( !(ulFlags & MMIO_CREATE) )
       {

        ulrc = GetAudioHeader (pInstance);

       } /* Not Create Flag */

    else
       {
       pInstance->ulDataSize = 0;
       }

    pInstance->fFileExists = TRUE;


    /*********************************************************************
    * You cannot do the set header immediately after file creation
    * because future sets on samples, bitpersample, channels may follow
   *********************************************************************/

    return (ulrc);

} /* OpenFile */
```

Networking Functions

Some OS/2 networks have specific IOProcs that can manage network traffic. The MMPM2.INI file contains information that describes the streaming quality when files are played or recorded over the network. The following example shows how to retrieve Quality of Service (QOS) values from the INI file.

If you are writing a streaming MCD, it is advantageous to use the MMIOM_BEGINSTREAM and MMIOM_ENDSTREAM messages to improve performance across a LAN. The following example uses the MMIOM_BEGINSTREAM message to inform the IOProcs that we want to stream across the network.

```
/* ------------------------------------------
* The MMPM2.INI file contains two variables that
* a streaming MCD should retrieve.  The first one
* QOS_VALUE (Quality of Service) contains settings
* which describe the quality of service that the
```

```
* network the user is streaming from will try to
* support (for example, GUARANTEED or DONTCARE).
* If this quality of service is not available, then another
* variable (QOSERRORFLAG) describes whether or not
* to notify the caller.
*-------------------------------------------------*/


ulrc = mciQuerySysValue( MSV_SYSQOSVALUE, &ulpInstance->lQosValue );

if ( !ulrc )
   {
   ulpInstance->lQosValue = DONTRESERVE;
   }

ulrc = mciQuerySysValue( MSV_SYSQOSERRORFLAG, &ulpInstance->lQOSReporting );

if ( !ulrc )
   {
   ulpInstance->lQOSReporting = ERROR_DEFAULT;
   }
```

The following example illustrates the EndQualityofService subroutine. Some OS/2 networks have specific IOProcs which can manage network traffic. This example uses the MMIOM_BEGINSTREAM and MMIOM_ENDSTREAM messages to inform these IOProcs that the waveaudio MCD wants to stream across the network.

```
ULONG   EndQualityofService ( INSTANCE      *pInstance )

{
LONG         rc;

  rc = mmioSendMessage( pInstance->hmmio,
                        MMIOM_ENDSTREAM,
                        0,
                        0 );

  return ( rc );

} /* EndQualityofService */
```

Using MMIO Editing Functions

This section shows the MMIO editing functions used by an MCD to implement media control interface editing functions. The MMIO functions and their media control interface equivalents are shown in the following table:

MMIO Functions	MCI Equivalents
MMIOM_DELETE	MCI_CUT
MMIOM_READ	MCI_COPY
MMIOM_DELETE	MCI_DELETE
MMIOM_UNDO	MCI_UNDO
MMIOM_REDO	MCI_REDO
MMIOM_DELETE, MMIOM_BEGININSERT, and MMIOM_ENDINSERT	MCI_PASTE

The following example shows how to use MMIO functions to paste data into a file.

```
/****************************************
* Remove the information in the file if
* from/to are specified.
****************************************/

if ( ulParam1 & MCI_FROM || ulParam1 & MCI_TO )
   {
   lReturnCode = mmioSendMessage( pInstance->hmmio,
                                  MMIOM_DELETE,
                                  pEditParms->ulFrom,
                                  ulPasteLen );

   if ( lReturnCode != MMIO_SUCCESS )
      {
      ulrc = mmioGetLastError( pInstance->hmmio );
      PasteNotify( &FuncBlock, ulParam1, ulrc );
      return ( ulrc );
      }
   }



if ( !( ulParam1 & MCI_TO_BUFFER ) )
   {
   /****************************************
   * Let the IOProc know that the information is
   * about to be inserted into the file
   ****************************************/

   lReturnCode = mmioSendMessage( pInstance->hmmio,
                                  MMIOM_BEGININSERT,
                                  0,
                                  0 );

   if ( lReturnCode != MMIO_SUCCESS )
      {
      ulrc = mmioGetLastError( pInstance->hmmio );
      PasteNotify( &FuncBlock, ulParam1, ulrc );
      return ( ulrc );
      }

   /****************************************
   * Write the information that we received from
   * the clipboard
   ****************************************/


   lReturnCode = mmioWrite( pInstance->hmmio,
                            ( PSZ ) pBuffer,
                            ulBuffLen );

   if ( lReturnCode == MMIO_ERROR )
      {
      ulrc = mmioGetLastError( pInstance->hmmio );
      PasteNotify( &FuncBlock, ulParam1, ulrc );
      return ( ulrc );
      }

   /****************************************
   * We have finished inserting the information
   * the paste is complete
   ****************************************/


  lReturnCode = mmioSendMessage( pInstance->hmmio,
                                 MMIOM_ENDINSERT,
                                 0,
                                 0 );

  if ( lReturnCode != MMIO_SUCCESS )
     {
     ulrc = mmioGetLastError( pInstance->hmmio );
     PasteNotify( &FuncBlock, ulParam1, ulrc );
     return ( ulrc );
     }
```

Sync/Stream Operations

The waveform audio MCD uses SPI functions to communicate with the Sync/Stream Manager (SSM) to prepare for data streaming, and to stream the data from the source location (the buffer) to the target location (the adapter).

Preparing to Stream the Waveform Data

When the waveform audio MCD receives a request to open a device, it issues SpiGetHandler and passes the names of the source and target stream handlers to the Sync/Stream Manager (SSM), so that it knows which stream handlers are going to stream the data. Data streaming cannot begin until the SSM has this information.

```
/*******************************************************
* Get 'A' stream Handler Handles for Source & target operations
* The file system stream handler is the default 'A' handler
* but the memory stream handler will be used for playlists.
*******************************************************/


   if (ulrc = SpiGetHandler((PSZ)DEFAULT_SOURCE_HANDLER_NAME,
                            &hidASource,
                            &hidATarget))
      {
      return ( ulrc );
      }


/*********************************************************
* Get 'B' stream Handler Handles for Source & target operations
* The audio stream handler is considered the B stream handler
* since it will usually be the target.
*********************************************************/

ulrc = SpiGetHandler( (PSZ)DEFAULT_TARGET_HANDLER_NAME,
                      &hidBSource,
                      &hidBTarget);

return ( ulrc );
```

After the waveform audio MCD gets its handler IDs, it must fill in the SPCBKEY data structure to tell the SSM what kind of data it is going to stream. SPCBKEY consists of three fields; the data type and two subdata types. For example, it might tell the SSM, "I'm going to stream PCM data at 8 bits, 22 kHz."

Next, the waveform audio MCD must fill in the device control block (DCB) to tell the SSM which device is being streamed to (as shown in the following example. The DCB contains two essential items of information:

- ASCII string device name
- Device handle

The ASCII string device name is obtained from the INI file and passed down by MDM. The device handle is the system file number returned by the IOCtl when the open to the device was done from AUDIOIF.

```
SysInfo.ulItem       = MCI_SYSINFO_QUERY_NAMES;
SysInfo.usDeviceType  = LOUSHORT(ulDeviceType);
SysInfo.pSysInfoParm = &QueryNameParm;

itoa (HIUSHORT(ulDeviceType), szIndex, 10);

szIndex[1] = '\0';

strncat (szAmpMix, szIndex, 2);
strcpy (QueryNameParm.szLogicalName, szAmpMix);
```

```
   if (rc = mciSendCommand (0,
                            MCI_SYSINFO,
                            MCI_SYSINFO_ITEM | MCI_WAIT,
                            (PVOID) &SysInfo,
                            0))
           return (rc);



   /***************************************
   * Get PDD associated with our AmpMixer
   * Device name is in pSysInfoParm->szPDDName
   ***************************************/

   SysInfo.ulItem       = MCI_SYSINFO_QUERY_DRIVER;
   SysInfo.usDeviceType  = (USHORT) ulDeviceType;
   SysInfo.pSysInfoParm = &SysInfoParm;

   strcpy (SysInfoParm.szInstallName, QueryNameParm.szInstallName);

   if (rc = mciSendCommand (0,
                            MCI_SYSINFO,
                            MCI_SYSINFO_ITEM | MCI_WAIT,
                            (PVOID) &SysInfo,
                            0))
       return (rc);

   strcpy (szPDDName, SysInfoParm.szPDDName);

   return ( MCIERR_SUCCESS );

} /* GetPDDName */
```

Creating the Stream

After handler IDs are obtained and the SPCBKEY and DCB information is filled in, the MCD makes the call to SpiCreateStream. At this point the two handlers are initiated to create their stream, buffers are allocated, and the stage is set for streaming.

The following example code illustrates how to create the stream. The caller supplies the source and target stream handlers and the audio device control block should have been filled in previously (see MCIOPEN.C). The caller also supplies the EventProc(edure) where all of the stream events will be reported.

```
  ulrc = SpiCreateStream ( hidSrc,
                           hidTgt,
                           &pInstance->StreamInfo.SpcbKey,
                           (PDCB) &pInstance->StreamInfo.AudioDCB,
                           (PDCB) &pInstance->StreamInfo.AudioDCB,
                           (PIMPL_EVCB) &pInstance->StreamInfo.Evcb,
                           (PEVFN) EventProc,
                           (ULONG) NULL,
                           hStream,
                           &pInstance->StreamInfo.hEvent);
```

When the MCD creates a stream, it must register a callback handle with SSM. The callback handle is an entry point or function within the MCD code that processes events coming back from SSM during the streaming process. The callback handle is a powerful mechanism because it frees the driver to do other work during the streaming process. When an event occurs, SSM detects it and reports it to the callback address.

The waveform audio MCD sample includes the following event routines:

- RecordEventRoutine (ADMCRECD.C) as shown in the following example.
- PlayEventRoutine (ADMCPLAY.C) as shown in the next example.

```
RC APIENTRY RecordEventRoutine ( MEVCB  *pevcb)
{
  MTIME_EVCB        *pMTimeEVCB;    // Modified EVCB
  INSTANCE          *ulpInstance;   // Instance Ptr

  /*************************************************************
  * EventProc receives asynchronous SSM event notifications
  * When the event is received, the event semaphore is posted
  * which will wake up the MCD thread(s) blocked on this
  * semaphore.
  * The semaphore is not posted for time events like
  * cuepoint (TIME) and media position changes since they do
  * not alter the state of the stream.
  *************************************************************/


  switch (pevcb->evcb.ulType)
  {
  case EVENT_IMPLICIT_TYPE:

       /* Retrieve our instance from the EVCB */

       ulpInstance = (INSTANCE *)pevcb->ulpInstance;

       switch (pevcb->evcb.ulSubType)
       {
       case EVENT_ERROR:
            ulpInstance->StreamEvent = EVENT_ERROR;

            /*************************************
            * Check for playlist specific error first
            *************************************/

            /************************************
            * End of PlayList event is received
            * as an implicit error event. It
            * is treated as a normal EOS
            ************************************/
            if (ulpInstance->usPlayLstStrm == TRUE)
                if (pevcb->evcb.ulStatus == ERROR_END_OF_PLAYLIST)
                    ulpInstance->StreamInfo.Evcb.evcb.ulStatus =
                        MMIOERR_CANNOTWRITE;

            DosPostEventSem (ulpInstance->hEventSem);
           break;

       case EVENT_STREAM_STOPPED:
            /*************************************
            * Event Stream Stopped. Release the
            * Blocked thread
            *************************************/
            ulpInstance->StreamEvent = EVENT_STREAM_STOPPED;
            DosPostEventSem (ulpInstance->hEventSem);
           break;

       case EVENT_SYNC_PREROLLED:
            /*****************************************
            * This event is received in response to a
            * preroll start. A Preroll start is done
            * on an MCI_CUE message.
            *****************************************/
            ulpInstance->StreamEvent = EVENT_SYNC_PREROLLED;
            DosPostEventSem (ulpInstance->hEventSem);
           break;
.
.
.
```

The MEVCB structure in the RecordEventRoutine is a modified IMPL_EVCB which SSM uses to report events (see the *OS/2 Multimedia Programming Reference*). You can add additional fields to extend the EVCB structure. For example, MEVCB shown in the following example adds an instance pointer containing local instance data. When the Sync/Stream Manager calls the waveform audio MCD, it allows the MCD to have access to the local instance data, which is tied to that event.

The PlayEventRoutine shown in the following example is presumed to receive all types of event notifications from SSM. The types include implicit events and cue point notifications in terms of both time and data. In response to cue point notifications a MCI_CUEPOINT message is returned to MDM by way of mdmDriverNotify.

```
RC APIENTRY PlayEventRoutine ( MEVCB     *pevcb)
{
  MTIME_EVCB        *pMTimeEVCB;      /* Modified Time EVCB*/
  INSTANCE          * ulpInstance;    /* Current Instance*/
  HWND              hWnd;             /* Callback Handle */
  BOOL              fPlayListDone = FALSE;

  /*********************************************************
  * EventProc receives asynchronous SSM event notifications
  * When the event is received, the event semaphore is posted
  * which will wake up the MCD thread(s) blocked on this
  * semaphore.
  * The semaphore is not posted for time events like
  * cuepoint (TIME) and media position changes since they do
  * not alter the state of the stream.
  *********************************************************/

  switch (pevcb->evcb.ulType)
  {
  case EVENT_IMPLICIT_TYPE:

       /* Retrieve our instance from the EVCB */

       ulpInstance = (INSTANCE *)pevcb->ulpInstance;

       /* Retrieve the callback handle to post messages on */

       hWnd = ulpInstance->hwndCallBack;

       switch (pevcb->evcb.ulSubType)
       {
       case EVENT_EOS:
            ulpInstance->StreamEvent = EVENT_EOS;
            DosPostEventSem (ulpInstance->hEventSem);
           break;

       case EVENT_STREAM_STOPPED:
            /* Self explanatory--someone stopped the stream */

            ulpInstance->StreamEvent = EVENT_STREAM_STOPPED;
            DosPostEventSem (ulpInstance->hEventSem);
           break;

       case EVENT_SYNC_PREROLLED:
            /***************************************
            * This event is received in response to a
            * preroll start. A Preroll start is done
            * on an MCI_CUE message.
            ***************************************/

            ulpInstance->StreamEvent = EVENT_SYNC_PREROLLED;
            DosPostEventSem (ulpInstance->hEventSem);
           break;

       case EVENT_PLAYLISTMESSAGE:

            /***************************************
            * We can receive this event if a playlist
            * parser hits the MESSAGE COMMAND.
            * NOTE: The MCD should return this message
            * with the callback handle specified on the
            * open.  This could be the source of much
            * grief if you return on the wrong handle.
            ***************************************/

            mdmDriverNotify ( ulpInstance->usWaveDeviceID,
                           ulpInstance->hwndOpenCallBack,
                           MM_MCIPLAYLISTMESSAGE,
                           (USHORT) MAKEULONG(pevcb->evcb.ulStatus,
                               ulpInstance->usWaveDeviceID),
                           (ULONG) pevcb->evcb.unused1);
           break;

       case EVENT_PLAYLISTCUEPOINT:

            /***********************************************
            * We can receive this event if a playlist
```

```
            * parser hits the CUEPOINT COMMAND opcode
            * in the playlist.  This differs from a "normal"
            * cuepoint because it is detected by the source,
            * rather than the target stream handler.
            ********************************************/


            mdmDriverNotify ( ulpInstance->usWaveDeviceID,
                              ulpInstance->hwndOpenCallBack,
                              MM_MCICUEPOINT,
                              (USHORT) MAKEULONG(pevcb->evcb.ulStatus,
                                  ulpInstance->usWaveDeviceID),
                              (ULONG) pevcb->evcb.unused1);
          break;


       } /* SubType case of Implicit Events */
      break;

  case EVENT_CUE_TIME_PAUSE:
       {
       /************************************************
       * This event will arrive if we played to a certain
       * position in the stream.  Let the play thread know
       * that we have reached the desired point.
       ************************************************/

       pMTimeEVCB = (MTIME_EVCB *)pevcb;
       ulpInstance = (INSTANCE *)pMTimeEVCB->ulpInstance;
       ulpInstance->StreamEvent = EVENT_CUE_TIME_PAUSE;

       DosPostEventSem (ulpInstance->hEventSem);
       }
       break;

  case EVENT_CUE_TIME:

       break;

  } /* All Events case */

  return (MCIERR_SUCCESS);

} /* PlayEventProc */
```

Events reported by SSM can be normal occurrences, such as an end-of-stream because playback is complete, or the events can be abnormal occurrences, such as an error returned during the streaming process. The driver needs to know about these events.

SSM reports two types of events: *implicit* and *explicit*. *Implicit* events are always reported. When one occurs, the driver must receive the notification of its occurrence from SSM, however, it does not have to take any action.

An *explicit* event is reported to the driver only when the driver requests to be notified of the event's occurrence. For example, the driver requests cuepoint notifications.

Event Processing

Before the stream is created, you can enable event notification for implicit and explicit events using the SpiEnableEvent function. The following example specifies the EVENT_CUE_TIME_PAUSE flag, which will cause the stream to be paused when the cuepoint is reached. When the stream reaches the event during a play or record, the audio stream handler signals the MCD. Note that the stream is only paused (not stopped).

```
RC CreateToEvent (INSTANCE *ulpInstance, ULONG ulTo)
{
/* rename this function CreateToEvent */


  ULONG ulrc;

  /*****************************************************
```

```
  * Set up a cue time pause event at the place in
  * the stream where the caller wants us to play/record
  * to.  Note: this event will pause the stream and
  * will be considerably more accurate than just
  * setting a cue point, receiving the event and stopping
  * the stream (since a normal cue point will force
  * bleed over).
  ****************************************************/

  ulpInstance->StreamInfo.TimeEvcb.hwndCallback
       = ulpInstance->hwndCallBack;
  ulpInstance->StreamInfo.TimeEvcb.usDeviceID
       = ulpInstance->usWaveDeviceID;
  ulpInstance->StreamInfo.TimeEvcb.evcb.ulType
      = EVENT_CUE_TIME_PAUSE;
  ulpInstance->StreamInfo.TimeEvcb.evcb.ulFlags
      = EVENT_SINGLE;
  ulpInstance->StreamInfo.TimeEvcb.evcb.hstream
      = ulpInstance->StreamInfo.hStream;
  ulpInstance->StreamInfo.TimeEvcb.ulpInstance
      = (ULONG) ulpInstance;
  ulpInstance->StreamInfo.TimeEvcb.evcb.mmtimeStream = ulTo;

  /* Enable the cue time pause event. */

  ulrc = SpiEnableEvent((PEVCB) &ulpInstance->StreamInfo.TimeEvcb.evcb,
                        (PHEVENT) &ulpInstance->StreamInfo.hPlayToEvent);

  return ( ulrc );

} /* CreateToEvent */
```

If you enable event notification for a particular event, it is equally important to remove the event handle for the event so it is not used by subsequent commands. When any given event is reported, it must be removed explicitly using the SpiDisableEvent function. Once an event is removed from the system, the event no longer is detected or reported to the application or MCD.

```
     SpiDisableEvent(ulpInstance->StreamInfo.hPlayToEvent);
```

Associating a Stream

After a stream is created, and before it is possible to start the stream, it is necessary to make sure that a data resource (or stream object) is identified for use with the stream. The following example shows an example of a stream being associated with an MMIO file handle. The file system stream handler (FSSH) will always be the stream handler that we want to associate the data object with, therefore, if we have created a playback stream then FSSH is the source, so associate with the source. On a record stream, FSSH is the target, so associate with the target.

```
  if (Operation == PLAY_STREAM)
      {

      ulrc = SpiAssociate ( (HSTREAM)*hStream,
                            hidSrc,
                            (PVOID) &pInstance->StreamInfo.acbmmio);

      }  /* Associating play stream */
```

Reassociating the Stream on MCI_LOAD

Typically, a stream is destroyed, recreated, and then associated with a new file. However, you can bypass these steps and reassociate a

new file with an existing stream as long as the file is of the same data type as the stream was created to handle. For example, in the case of waveaudio, if you had a 16-bit 11KB WAVE stream, the file you want to associate with the stream must be of the same data type and sub type.

Note that only one data object may be associated with a stream once the stream is created, and it is specific to a particular stream handler (source or target). Associations may be changed, but the stream cannot be active; it must be stopped first (discard stop, flush stop, or EOS).

```
/*********************************************
* Reassociate The Stream Handlers with the new
* stream object if the stream has been created
* in the correct direction already.
*********************************************/
if (ulpInstance->ulCreateFlag == PREROLL_STATE)
    {
    /*****************************************
    * Fill in Associate Control Block Info for
    * file system stream handler (FSSH).  FSSH will
    * use the mmio handle we are associating to
    * stream information.
    *****************************************/
    ulpInstance->StreamInfo.acbmmio.ulObjType = ACBTYPE_MMIO;
    ulpInstance->StreamInfo.acbmmio.ulACBLen = sizeof (ACB_MMIO);
    ulpInstance->StreamInfo.acbmmio.hmmio = ulpInstance->hmmio;

    /*********************************************
    * Associate FileSystem as source if Playing. Note
    * the association is always done with the file
    * system stream handler since it is involved with
    * mmio operations.  If you try this with the
    * audio stream handler, you will get invalid
    * handle back.
    *********************************************/

    if (AMPMIX.ulOperation == OPERATION_PLAY)
       {
       ulrc = SpiAssociate ( ulpInstance->StreamInfo.hStream,
                             ulpInstance->StreamInfo.hidASource,
                             (PVOID) &ulpInstance->StreamInfo.acbmmio );
       }
    /*********************************************
    * Associate FileSystem as target if recording
    *********************************************/

    else
       {
       ulrc = SpiAssociate ( ulpInstance->StreamInfo.hStream,
                             ulpInstance->StreamInfo.hidATarget,
                             (PVOID) &ulpInstance->StreamInfo.acbmmio );

       } /* else we are in record mode */
```

Prerolling the Stream - Performance Considerations

Prerolling a stream allows the source stream handlers to start and fill the buffers. An application can then start the streams for better real-time response and initial synchronization of streams. This is accomplished by the SpiStartStream function with the SPI_START_PREROLL flag.

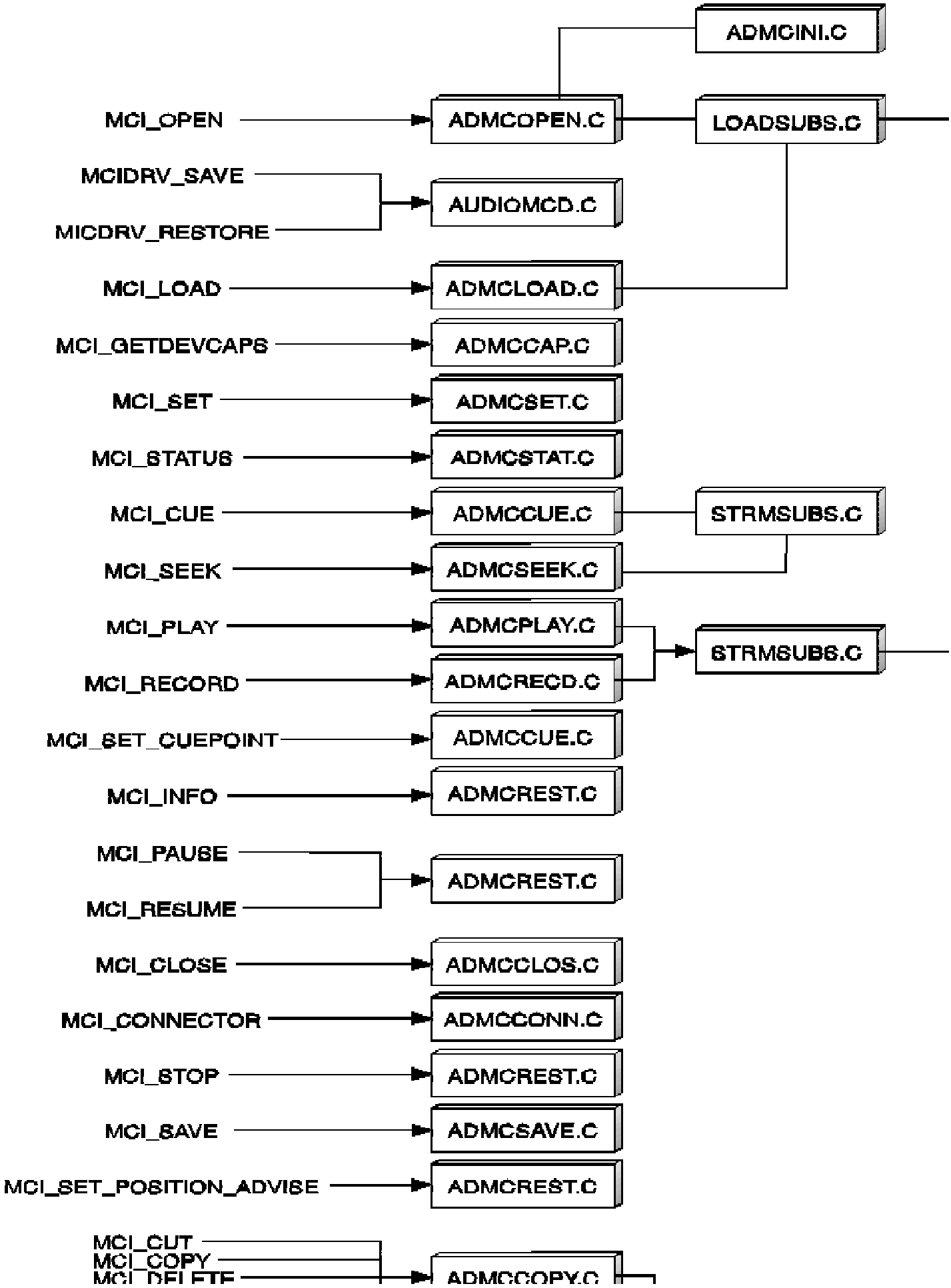
However, if a stream is already paused (STOP_PAUSED), the stream is already cued so there is no need to call SpiStartStream to refill the buffers. The following example illustrates how to check to see if a stream is in a paused state.

```
  /***********************************
  * If the stream is paused, there is no
  * sense in cueing it since the buffers
  * are full anyway
  ***********************************/

  if ( STRMSTATE == MCI_PAUSE  ||
```

```
 STRMSTATE == STOP_PAUSED )

{
/*********************************************
** If the stream is going the right way
** then we have the ability to avoid the cue
** since the buffers have been filled up before
** we did the pause
*********************************************/

if ( AMPMIX.ulOperation == OPERATION_RECORD &&
     fCueInput )
   {
   ulpInstance->ulCreateFlag = PREROLL_STATE;
   STRMSTATE = CUERECD_STATE;
   return ( MCIERR_SUCCESS );
   }

/*********************************************
* If the current stream is cued for playback and
* we have a cue_output request, our work is done
*********************************************/

else if ( AMPMIX.ulOperation == OPERATION_PLAY &&
          fCueOutput )
   {

   ulpInstance->ulCreateFlag = PREROLL_STATE;
   STRMSTATE = CUEPLAY_STATE;
   return ( MCIERR_SUCCESS );
   }

}   /* If the stream may be in cue state */
```

Waveform Audio MCD Modules

The following figure illustrates the outline of the waveform audio media control driver (AUDIOMCT.DLL).


ADMCINI.C
MCI_OPEN
ADMCOPEN.C
LOADSUBS.C
MCIDRV_SAVE
MICDRV_RESTORE
AUDIOMCD.C
MCI_LOAD
ADMCLOAD.C
MCI_GETDEVCAPS
ADMCCAP.C
MCI_SET
ADMCSET.C
MCI_STATUS
ADMCSTAT.C
MCI_CUE
ADMCCUE.C
STRMSUBS.C
MCI_SEEK
ADMCSEEK.C
MCI_PLAY
ADMCPLAY.C
STRMSUB6.C
MCI_RECORD
ADMCRECD.C
MCI_SET_CUEPOINT
ADMCCUE.C
MCI_INFO
ADMCREST.C
MCI_PAUSE
MCI_RESUME
ADMCREST.C
MCI_CLOSE
ADMCCLOS.C
MCI_CONNECTOR
ADMCCONN.C
MCI_STOP
ADMCREST.C
MCI_SAVE
ADMCSAVE.C
MCI_SET_POSITION_ADVISE
ADMCREST.C
MCI_CUT
MCI_COPY
MCI_DELETE
ADMCCOPY.C

Waveform Audio Media Control Driver DLL (AUDIOMCT.DLL)

Following describes the files and messages that are processed in in the AUDIOMCT.DLL.

File	Description
ADMCOPEN.C	Processes the MCI_OPEN message. On an MCI_OPEN, a streaming MCD should perform the following actions: 1. Check flags and validate pointers. 2. Get default values from the INI files (if necessary). 3. Process MCI_OPEN_PLAYLIST (if supported). 4. Process MCI_OPEN_MMIO (if supported). 5. Process MCI_OPEN_ELEMENT (if supported). 6. Connect to the amp/mixer. 7. Get the stream protocol key (if necessary).
LOADSUBS.C	Provides various utility functions used by MCI_OPEN and MCI_LOAD including: 1. Creating temporary filenames (CheckForValidElement) 2. Aborting in process commands (LoadAbortNotify) 3. Processing the OPEN_MMIO flag (OpenHandle) 4. Rational as to when to open the card in record or playback mode (OpenHandle) (ProcessElement). 5. Processing temp files (ProcessElement). 6. Opening a file with MMIO (ProcessElement). 7. Processing the MCI_READ_ONLY flag (ProcessElement). 8. Creating a temporary file (SetupTempFiles). 9. Using mmioSendMessage API to talk to an IOProc (SetupTempFiles). 10.Retrieving a connection and opening the connected device. 11.Stream Handler Setup (StreamSetup) 12.Processing MCI_NOTIFY or MCI_WAIT and callback handles (NotifyWaitSet up).
AUDIOMCD.C	Processes the MCIDRV_RESTORE message. A streaming MCD will receive a restore message when it regains control of the device they are attached to (for example, someone quit another application causing us to gain use of the device). On a restore, the MCD should check to see if it is in a paused state. If so, resume the stream. Processes the MCIDRV_SAVE message. A streaming MCD will receive a save message when it loses control of the device it is attached to (for example, someone started another application which takes over the waveaudio device). On a save, the MCD should check to see if the MCD is currently streaming (either record or playback). If so, pause the stream, and set a flag stating that the MCD was saved.
ADMCLOAD.C	Processes the MCI_LOAD message. This file illustrates the following concepts: 1. How to check flags on a load. 2. How to stop any commands which are in process. 3. Why cuepoints/positionadvises need to be turned off on a MCI_LOAD. 4. Handling OPEN_MMIO on an MCI_LOAD. 5. Why reassociation of a stream on MCI_LOAD is desirable and when it is appropriate.
ADMCCAP.C	Processes the MCI_GETDEVCAPS message. This file illustrates several concepts used to handle MCI_GETDEVCAPS including: 1. How to process the capability message commands. These are messages which this MCD supports (such as play, close etc.) Messages (or commands to the caller) such as sysinfo are not supported by this MCD. 2. How to process the capability item flag. Items describe particular features (such as the ability to record) which the MCD either does or does not support.
ADMCSTAT.C	Processes the MCI_STATUS message. This file illustrates several concepts used to handle MCI_STATUS including: 1. When/If to report media position within a stream. 2. How to determine the length of an existing file. 3. How to determine the length of a file which is currently being recorded.

4. Communicating with the amp/mixer to determine volume,
and other amplifier specific commands.
5. Reporting our current mode.
6. Reporting the current time format.

ADMCCUE.C

Processes the MCI_CUE message. Applications will typically call MCI_CUE in order to reduce the time required to begin the initial record or play. To a streaming MCD, MCI_CUE translates to an SPI_START_PREROLL. The start preroll will fill up all of the initial streaming buffers but will not start the stream. Then, when MCI_PLAY or MCI_RECORD are called, when they do an SpiStartStream then stream can start immediately.

On an MCI_CUE, a streaming MCD should perform the following actions:
1. Check flags and validate pointers.
2. Check to see if the stream is already in cue state. If it is, then
just return success.
3. If the caller wants to cue for output, execute the following:
a. stop any in process commands.
b. If the stream is going in the wrong direction (that is, record),
destroy it.
c. Create the stream if necessary.
d. Preroll start the stream.

4. If the caller wants to cue for input, execute the following:
a. Stop any in process commands.
b. If the stream is going in the wrong direction, destroy it.
c. Create the stream if necessary.
d. Preroll start the stream.

Processes the MCI_SET_CUEPOINT message. On a MCI_SET_CUEPOINT, a streaming MCD should perform the following actions:
1. Check flags and validate pointers.
2. Turn the cuepoint on or off depending on what flags are passed in.

ADMCSEEK.C

Processes the MCI_SEEK message. On a seek, a streaming MCD should perform the following actions:
1. Verify that the flags passed are valid.
2. Verify the MCI_FROM, MCI_TO parameter if they were passed in.
3. Ensure that any pointers passed are valid.
4. Stop any commands which are active on another thread.
5. If no stream has been created, then create one.
6. If a stream had previously been created, ensure that it is in
stopped state.

ADMCPLAY.C

Processes the MCI_PLAY message. On a MCI_PLAY, a streaming MCD should perform the following actions:
1. Always check flags and validate memory first. This way,
if the flags are invalid, the previous command will not be interrupted.
2. If there is a command active on another thread (that is, a play,
record or save), then either abort (record or save) or supersede
(play) by stopping the stream and sending a message to the caller.
3. If the stream is going the wrong way (for example,
it is set up for recording) then destroy the stream.
4. If no stream has been created, then create one. If the stream handler
needs to associate a data object, do it here.
5. If we destroyed a recording stream before creating the play back stream,
ensure that playback stream has the same position as the previous
record stream.
6. Enable any events (such as cuepoints or position advises).
7. Start stream.
8. Wait for a streaming event.
9. Stop the stream if necessary.
10.If MCI_NOTIFY was sent used, inform the caller of command completion.

ADMCRECD.C

Processes the MCI_RECORD message. On a MCI_RECORD, a streaming MCD should perform the following actions:
1. Always check flags and validate memory first. This way, if the flags
are invalid, the previous command will not be interrupted.
2. If there is a command active on another thread (that is,
a play, record or save), then either abort (play or save)
or supersede (record) by stopping the stream and sending
a message to the caller.
3. If the stream is going the wrong way (for example,
it is setup for playback) then destroy the stream.

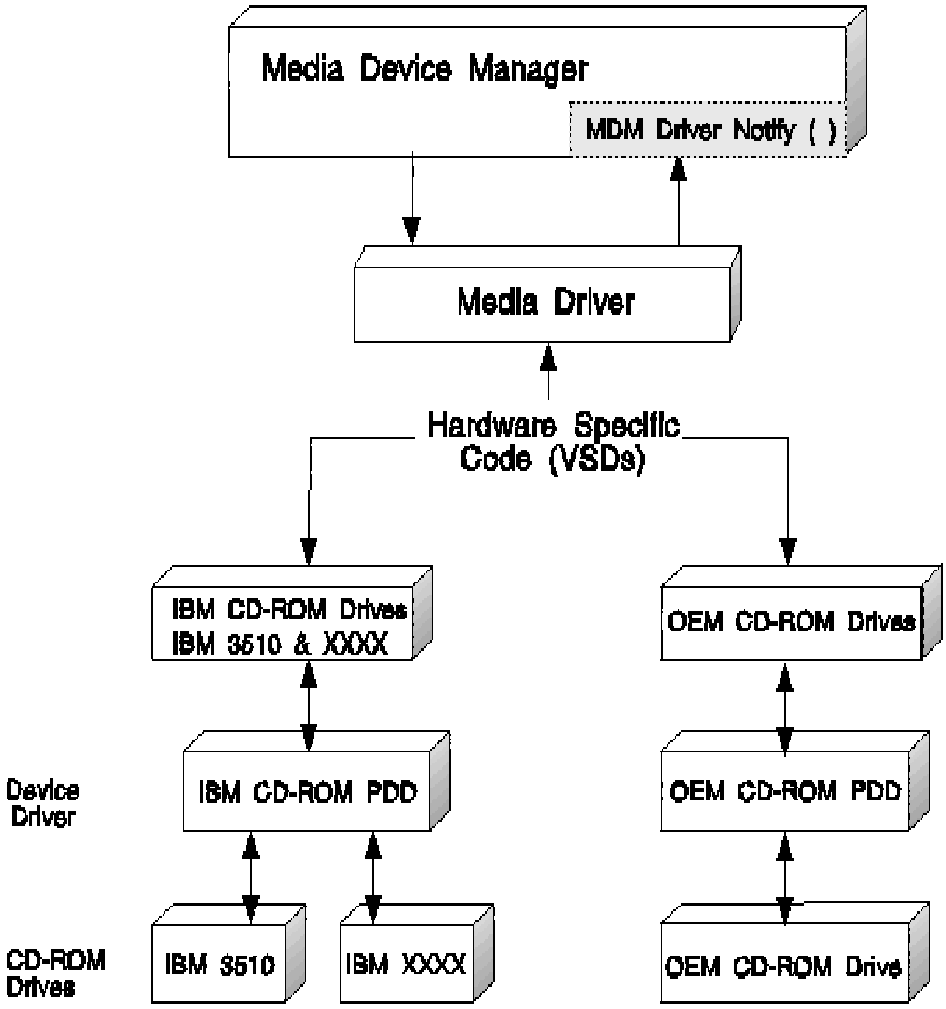
	4. If no stream has been created, then create one. If the stream handler needs to associate a data object, do it here. 5. If we destroyed a play back stream before creating the record stream, seek to the same position in the record stream where the play back stream was. 6. Enable any events (such as cuepoints or position advises). 7. Start stream. 8. Wait for a streaming event. 9. Stop the stream if necessary. 10.If MCI_NOTIFY was sent used, inform the caller of command completion.
ADMCCLOS.C	Processes the MCI_CLOSE message. On a close, a streaming MCD must perform the following actions: 1. Stop all commands which are active on another thread(s). 2. Destroy all active streams. 3. Close all open files. 4. Delete any temporary files. 5. Close any connected devices (such as an amp-mixer). 6. If MCI_NOTIFY was used, notify the caller of completion.
ADMCCONN.C	Processes the MCI_CONNECTOR message. This source file illustrates how to enable, disable and query connectors. It also illustrates how to pass messages on to a connected MCD.
ADMCREST.C	Processes the MCI_STOP message. On a stop, a streaming MCD should perform the following actions: 1. Verify that the flags passed are valid. 2. Stop any commands which are active on another thread. 3. If a stream had previously been created, ensure that it is in stopped state. 4. If it is a paused stream, then do a STOP_PAUSE to ensure that no data will be lost. Processes the MCI_RESUME message. On a MCI_RESUME, a streaming MCD should perform the following actions: 1. Ensure that no flags are passed in. 2. If we are paused, resume the stream. Processes the MCI_SETPOSITIONADVISE message. To a streaming MCD, a position advise simply is a cuepoint which reoccurs every x time units. To enable position advise, the following steps should be followed: 1. Check flags and validate and pointers. 2. If the caller has asked for position advise to be turned on then a. If a stream has been created, then enable the recurring cuepoint event. b. Else, set a flag and enable the event later. 3. If position advise is to be turned off, then disable the recurring cuepoint event. Processes the MCI_PAUSE message. On a MCI_PAUSE, a streaming MCD should perform the following actions: 1. Ensure that no flags are passed in. 2. If we are currently streaming, pause the stream. 3. Set flag indicating that we are in paused state.
ADMCSAVE.C	Processes the MCI_SAVE message. MCI_SAVE is not a required command, however, if a streaming MCD takes advantage of temporary files, then it should use MCI_SAVE. 1. Ensure that the flags are valid. 2. Then ensure that all pointers point to valid memory. 3. If any operations are currently active, they are then aborted. 4. Finally, the save operation itself is then done.
AUDIOSUB.C	Contains various utility functions including: 1. Correct processing of notifications. 2. Handling calls the are neither MCI_WAIT or MCI_NOTIFY (PostMDMMessage). 3. Using mmioGetHeader to obtain audio settings from a file (GetHeader). 4. Creating a playlist stream (AssociatePlaylist). 5. Installing an IOProc (InstallIOProc). 6. Communicating with IOProcs with different capabilities (OpenFile). 7. Processing audio files with various file formats (OpenFile). 8. Time format conversion (ConvertToMM + ConvertTimeFormat). 9. Creating an SPI stream (CreateNAssociateStream) 10.Associating an SPI stream (CreateNAssociateStream)

	11.Setting an event for use in play to/record to (DoTillEvent).
ADMCPST.C	Processes the MCI_PASTE message. This file illustrates using MMIO functions and clipboard functions to: 1. Process the MCI_FROM_BUFFER flag or MCI_TO_BUFFER. 2. Handle default positioning if no MCI_FROM/MCI_TO flags are passed 3. Use MMIO to insert information into a file.
ADMCCOPY.C	Processes the MCI_COPY, MCI_DELETE, MCI_CUT, MCI_UNDO and MCI_REDO messages. It is a generic routine which handles placing information into the clipboard: 1. Default positioning for cut/copy/delete messages. 2. Using MMIOM_DELETE to remove information from a file. 3. Positioning after a cut/copy/delete 4. Using MMIOM_UNDO and MMIOM_REDO messages.
ADMCEDIT.C	Contains utilities used by clipboard functions (such as cut and copy). The file illustrates: 1. How to Process MCI_FROM / MCI_TO for edit operations (CheckEditFlags) . 2. Determine default from/to positions. 3. Generic stream abort routine 4. Retrieving information from the clipboard. 5. Placing information into the clipboard. 6. Using MMIO memory files to manipulate clipboard data.
ADMCINI.C	Illustrates how to parse device specific parms from the INI file.
STRMSUBS.C	Contains streaming subroutines used by play, record and cue. 1. How to close mmio files, and temp files (CloseFile). 2. How to stop a record/playback stream (StopStream). 3. How to use a semaphore to protect in progress commands from being aborted in sensitive areas. 4. Why a stream must be destroyed after an MCI_SET (DestroySetStream). 5. How to handle superseding and aborting of notifies. 6. How and why to set a cue time pause event for MCI_TO (ProcessFromToFlags). 7. How to enable cuepoints and positionchanges (EnableEvents). 8. Networking functions (BeginQualityOfService, EndQualityOfService)

Controlling a Nonstreaming Device: CD Audio MCD

The CD audio MCD controls a nonstreaming device: the CD-ROM drive. Because it is a nonstreaming device, the CD-ROM drive does not require buffered I/O for streaming by the sync/stream manager. Instead, it plays audio out through its internal Digital-to-Analog Converter (DAC), which is the nonstreaming device equivalent of an audio adapter. Thus, the CD audio MCD must use the IOCtl interface to process its own commands to manipulate the data stream within the device.

The modules of the CD audio MCD are shown in the following figure.



Setting Audio Attributes

Because the CD-ROM drive is a nonstreaming device, the CD audio MCD is not linked to the amp mixer. This means the driver must do its own processing of MCI_SET commands that set audio attributes such as volume changes, using the IOCtl interface.

Processing an MCI_PLAY Command

The CD audio MCD receives commands by means of the mciDriverEntry point, same as the waveform audio MCD. The command is passed to the vendor-specific driver (VSD) module.

To see how the CD audio MCD processes these commands, let's trace the processing of an MCI_PLAY command. To begin with, the vsdDriverEntry function makes a call to process_msg, passing the message and message parameters as shown in the following example.

```
ULONG APIENTRY vsdDriverEntry(PVOID pInstance, USHORT usMessage,
                              ULONG *pulParam1, PVOID pParam2,
                              USHORT usUserParm)
{
   ULONG  rc, ulP1Temp = MCI_WAIT;
   USHORT try = 1;
```

```
   if (pulParam1 == 0L)
      pulParam1 = &ulP1Temp;

   /* check to see if the drive is open, unless it is an Open message */
   if (usMessage == MCI_OPEN)
      rc = CDAudOpen(*pulParam1, (MMDRV_OPEN_PARMS *)pParam2);
   else
   {
      /* if the device is closed try reopening it unless you are closing */
      if (((PINST) lpInstance)->hDrive == 0 && usMessage != MCI_CLOSE)
      {
         rc = CD01_Open((PINST) lpInstance);
         /* Clear commands not needing an open hardware device */
         if (rc == MCIERR_DEVICE_NOT_READY)
            if ((usMessage == MCI_DEVICESETTINGS) ||
                (usMessage == MCI_GETDEVCAPS) ||
                (usMessage == MCI_INFO) ||
                (usMessage == MCIDRV_REGISTER_DRIVE) ||
                (usMessage == MCI_SET_CUEPOINT) ||
                (usMessage == MCI_SET_POSITION_ADVISE) ||
                (usMessage == MCIDRV_CD_STATUS_CVOL) ||
                (usMessage == MCIDRV_SYNC &&
                   !(*pulParam1 & MCIDRV_SYNC_REC_PULSE)))
               rc = MCIERR_SUCCESS;

      }  /* of if drive needs to be open */
      else      /* drive was opened */
         rc = MCIERR_SUCCESS;

      if (!rc)
         do
         {
            /* process message */
            rc = process_msg((PINST) lpInstance, usMessage,
                             pulParam1, pParam2, usUserParm);
            if (rc == MCIERR_DEVICE_NOT_READY ||       /* ERROR RECOVERY */
                rc == MCIERR_MEDIA_CHANGED)
            {
               if (((PINST)lpInstance)->Drive == '0') /* drive is closed */
               {                                      /* do not reissue commands */
                  rc = MCIERR_SUCCESS;
                  break;
               }
               else
                  if (try == 2)
                     break;                           /* quit after 2 tries. */
                  else
                  {
                     rc = CDAudErrRecov((PINST) lpInstance);
                     if (rc)                 /* error is still there, exit */
                        break;
                     else
                        try++;
                  }  /* of else only tried the command once (try == 1) */

            }  /* of if the drive was not ready */
            else
               break;                                 /* clear flag to exit */

         } while (try);  /* end of do loop and if no open error */

   } /* of else command was not MCI_OPEN */

   return(rc);

}  /* of vsdDriverEntry() */
```

The process_msg routine has a long case statement. The case in which we are interested is for MCI_PLAY. When process_msg calls CD01_Play, it passes the following information:

- Instance
- msgParam1
- FROM value received from msgParam2
- TO value received from msgParam2
- MCI_PLAY command.

```
static ULONG process_msg(PINST pInst, USHORT usMessage,
                         ULONG *pulParam1, PVOID pParam2, USHORT usUserParm)
{
   ULONG rc;

   DosRequestMutexSem(pInst->hInstSem, WAIT_FOREVER);
   if (usMessage != MCI_PLAY)
      DosReleaseMutexSem(pInst->hInstSem);       // No protection needed

   /* process message */
   switch(usMessage)
   {
      case MCI_CLOSE :
         rc = CDAudClose(pInst, *pulParam1);
         break;
      case MCI_CUE :                          /* Pre-roll */
         rc = CD01_Cue(pInst);
         break;
      case MCI_GETDEVCAPS :                   /* Get Device Capabilities */
         rc = CD01_GetCaps(*pulParam1, (MCI_GETDEVCAPS_PARMS *)pParam2);
         break;
      case MCI_GETTOC :                         /* Get Table of Contents */
         if (*pulParam1 & WAIT_NOTIFY_MASK)
            rc = MCIERR_INVALID_FLAG;
         else
            rc = CD01_GetTOC(pInst, (MCI_TOC_PARMS *)pParam2);
         break;
      case MCI_INFO :
         rc = CDAudInfo(pInst, *pulParam1, (MCI_INFO_PARMS *)pParam2);
         break;
      /* case MCI_OPEN :     open was already done in vsdDriverEntry() */

      case MCI_PAUSE :
         rc = CD01_Stop(pInst, TIMER_PLAY_SUSPEND);
         break;
      case MCI_PLAY :
         rc = CD01_Play(pInst, pulParam1,
                        ((MCI_PLAY_PARMS *)pParam2)->ulFrom,
                        ((MCI_PLAY_PARMS *)pParam2)->ulTo, usUserParm,
                        ((MCI_PLAY_PARMS *)pParam2)->hwndCallback);
         break;
      case MCIDRV_REGISTER_DISC :                /* Register Disc */
         rc = CDAudRegDisc(pInst, REG_BOTH, (MCI_CD_REGDISC_PARMS *)
           pParam2);
         break;
      case MCIDRV_REGISTER_DRIVE :               /* Register Drive */
         rc = CDAudRegDrive(pInst, (MCI_CD_REGDRIVE_PARMS *)pParam2);
         break;
      case MCIDRV_REGISTER_TRACKS :              /* Register Tracks */
         rc = CD01_RegTracks(pInst, (MCI_CD_REGTRACKS_PARMS *)pParam2);
         break;
      case MCIDRV_RESTORE :
         rc = CD01_Restore(pInst, (MCIDRV_CD_SAVE_PARMS *)pParam2);
         break;
      case MCI_RESUME :                          /* Unpause */
         rc = CD01_Resume(pInst);
         break;
      case MCIDRV_SAVE :
         rc = CD01_Save(pInst, (MCIDRV_CD_SAVE_PARMS *)pParam2);
         break;
      case MCI_SEEK :
         rc = CD01_Seek(pInst, ((MCI_SEEK_PARMS *)pParam2)->ulTo);
         break;
      case MCI_SET :
         rc = CDAudSet(pInst, pulParam1, (MCI_SET_PARMS *)pParam2);
         break;
      case MCI_SET_CUEPOINT :
         rc = CD01_CuePoint(pInst, *pulParam1, (MCI_CUEPOINT_PARMS *)
            pParam2);
         break;
      case MCI_SET_POSITION_ADVISE :
         rc = CD01_PosAdvise(pInst, *pulParam1, (MCI_POSITION_PARMS *)
            pParam2);
         break;
      case MCIDRV_CD_SET_VERIFY :
         rc = CDAudSetVerify(*pulParam1);
```

```
         break;
      case MCI_STATUS :
         rc = CDAudStatus(pInst, *pulParam1, (MCI_STATUS_PARMS *)pParam2);
         break;
      case MCIDRV_CD_STATUS_CVOL :
         rc = CDAudStatCVol(&((MCI_STATUS_PARMS *)pParam2)->ulReturn);
         break;
      case MCI_STOP :
         rc = CD01_Stop(pInst, TIMER_EXIT_ABORTED);
         break;
      case MCIDRV_SYNC :
         rc = CD01_Sync(pInst, *pulParam1, (MCIDRV_SYNC_PARMS *)pParam2);
         break;

      /* List unsupported functions */

      case MCI_ACQUIREDEVICE :  case MCI_CONNECTION :  case MCI_CONNECTOR :
      case MCI_CONNECTORINFO :  case MCI_DEVICESETTINGS :
      case MCI_DEFAULT_CONNECTION:                     case MCI_ESCAPE :
      case MCI_LOAD:            case MCI_MASTERAUDIO : case MCI_RECORD :
      case MCI_RELEASEDEVICE :  case MCI_SAVE :        case MCI_SPIN :
      case MCI_STEP :           case MCI_SYSINFO :     case MCI_UPDATE :
      case MCIDRV_CD_READ_LONG :
         rc = MCIERR_UNSUPPORTED_FUNCTION;
         break;
      default : rc = MCIERR_UNRECOGNIZED_COMMAND;

   }  /* of switch */

   return(rc);
```

When CD01_Play receives a PLAY command, processing goes to the line that issues the call to CallIOCtl.

```
ULONG CD01_Play(PINST pInst, ULONG *pulParam1, ULONG ulFrom, ULONG ulTo,
                USHORT usUserParm, HWND hwndCallback)
{
   ULONG rc;
   ULONG ulThreadID;
   ULONG cnt;

   /* Stop drive before issuing next play command */
   if ((pInst->usPlayFlag == TIMER_PLAYING) ||
       (pInst->usPlayFlag == TIMER_PLAY_SUSPEND) ||
       (pInst->usPlayFlag == TIMER_PLAY_SUSP_2))
      if (*pulParam1 & MCI_NOTIFY)
         CD01_Stop(pInst, TIMER_EXIT_SUPER);
      else
         CD01_Stop(pInst, TIMER_EXIT_ABORTED);

   /* prepare for play call */
   pInst->ulCurPos = ulFrom;
   pInst->ulEndPos = ulTo;
   pInst->usPlayNotify = (USHORT)(*pulParam1 & (MCI_WAIT | MCI_NOTIFY));
   if (*pulParam1 & MCI_NOTIFY)
   {
      pInst->usPlayUserParm = usUserParm;
      pInst->hwndPlayCallback = hwndCallback;
      *pulParam1 ^= MCI_NOTIFY;
   }  /* notify flag was used */

   if (*pulParam1 & MCI_WAIT)
      rc = CD01_Timer(pInst);      /* returns when play commands end */
   else
   {
      DosResetEventSem(pInst->hReturnSem, &cnt);  /*force a wait
      rc = DosCreateThread(&ulThreadID, (PFNTHREAD)CD01_Timer,
                           (ULONG)pInst, 0L, THREAD_STACK_SZ);
      if (rc)
      {
         rc = MCIERR_OUT_OF_MEMORY;
         DosPostEventSem(pInst->hReturnSem);
      }
      else  /* wait for new thread to process enough */
      {
         /* Let MCD know not to send notification by returning wait */
         *pulParam1 = (*pulParam1 & ~MCI_NOTIFY) | MCI_WAIT;
```

```
         /* wait for new thread to process enough */
         DosWaitEventSem(pInst->hReturnSem, WAIT_FOREVER);
      }

      DosReleaseMutexSem(pInst->hInstSem);

   }  /* else no wait flag was used */

   return(rc);

}  /* of CD01_Play() */

/*************************************************************************/
/*                                                                       */
/* SUBROUTINE NAME:  CD01_PlayCont                                       */
/*                                                                       */
/* DESCRIPTIVE NAME:  CD Play Continue.                                  */
/*                                                                       */
/* FUNCTION:  Continue to play audio data to internal DAC(s) from a      */
/*            MCIDRV_RESTORE or MCIDRV_SYNC command.                     */
/*                                                                       */
/* PARAMETERS:                                                           */
/*      PINST pInst       -- Instance pointer.                           */
/*      ULONG ulFrom      -- From address.                               */
/*      ULONG ulTo        -- To address in MMTIME.                       */
/*                                                                       */
/* EXIT CODES:                                                           */
/*      MCIERR_SUCCESS   -- action completed without error.              */
/*      MCIERR_DEVICE_NOT_READY  -- device was not ready, no disc.       */
/*      MCIERR_MEDIA_CHANGED     -- Disc changed.                        */
/*                                                                       */
/* NOTES:                                                                */
/*                                                                       */
/*************************************************************************/
ULONG CD01_PlayCont(PINST pInst, ULONG ulFrom, ULONG ulTo)
{
   ULONG rc;
   BYTE param[PLAYAUD_PMAX] = {'C', 'D', '0', '1', RBMODE};
   ULONG ulDataLen = STANDRD_DMAX, ulParamLen = PLAYAUD_PMAX;

   /* convert starting MM Time into Redbook 2 format */
   * (ULONG *)&param[STARTFFFLD] = REDBOOK2FROMMM(ulFrom);

   /* convert ending MM Time into Redbook 2 format */
   * (ULONG *)&param[END_FF_FLD] = REDBOOK2FROMMM(ulTo);

   /* Stop drive before issuing next play command */
   CD01_Stop(pInst, TIMER_PLAY_SUSPEND);

   /* play drive */
   rc = CallIOCtl(pInst, CDAUDIO_CAT, PLAY__AUDIO,
                  param, ulParamLen, &ulParamLen,
                  NULL,  ulDataLen,  &ulDataLen);
   if (!rc)
      pInst->ulCurPos = ulFrom;

   /* if Timer was stopped, continue timer loop */
   if (pInst->usPlayFlag == TIMER_PLAY_SUSPEND)
      pInst->usPlayFlag = TIMER_PLAYING;
   DosPostEventSem(pInst->hTimeLoopSem);

   return(rc);

}  /* of CD01_PlayCont() */
```

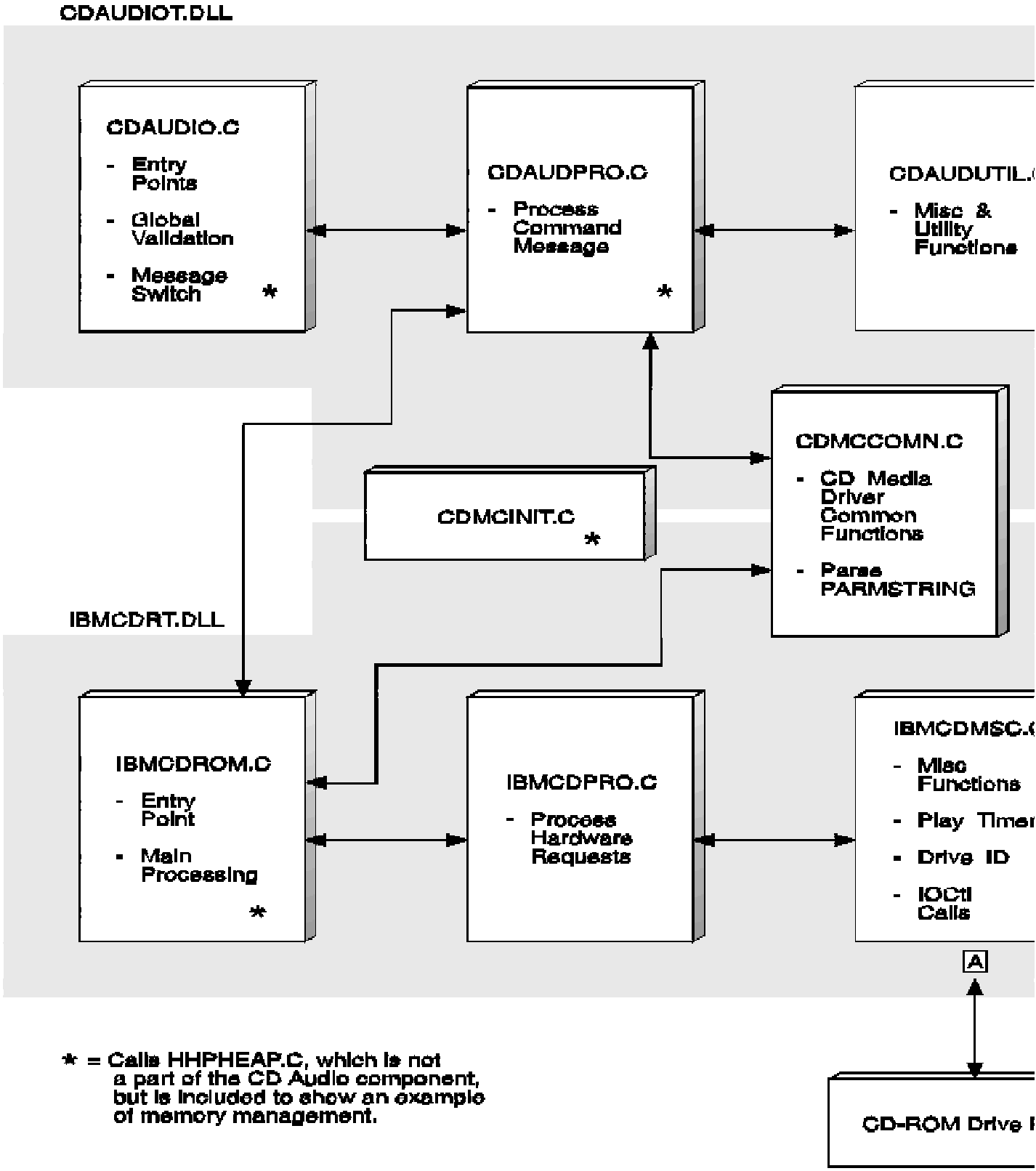
The CallIOCtl routine issues the DosDevIOCtl request, passing the following parameters: *hDrive* is the device handle retrieved from a DosOpen call, *ulCat* is the IOCtl category, and *ulFunction* is the function number. Also required are buffer pointers, buffer sizes and lengths, and pointers to the buffer lengths for the parameter and the data buffers. The Parameter buffer is used for input, and the data buffer is used for output. The pointers to buffer lengths enable the physical device driver to return the actual lengths of the buffers.

```
ULONG CallIOCtl(PINST pInst,  ULONG ulCat,  ULONG ulFunction,
                PVOID pParam, ULONG ulPLen, ULONG *pulPLen,
                PVOID pData,  ULONG ulDLen, ULONG *pulDLen)
{
   ULONG rc;
```

```
   DosRequestMutexSem(pInst->hIOSem, (ULONG)-1L);
   rc = DosDevIOCtl(pInst->hDrive, ulCat, ulFunction,
                    pParam, ulPLen, pulPLen,
                    pData,  ulDLen, pulDLen);
   DosReleaseMutexSem(pInst->hIOSem);

   switch(rc)
   {
      case 0L :
         rc = MCIERR_SUCCESS;
         break;
      case 0xFF03 :
         rc = MCIERR_UNSUPPORTED_FUNCTION;
         break;
      case 0xFF06 :
      case 0xFF08 :
         rc = MCIERR_OUTOFRANGE;
         break;
      case 0xFF04 :
      case 0xFF0C :
         rc = MCIERR_HARDWARE;
         break;
      case 0xFF10 :
         rc = MCIERR_MEDIA_CHANGED;
         break;
      default :
         rc = MCIERR_DEVICE_NOT_READY;
   }

   return(rc);

}  /* of CallIOCtl() */
```

CD Audio MCD Modules

The following figure illustrates the outline of the CD audio MCD (CDAUDIOT.DLL) and the vendor-specific driver (VSD) for the IBM 3510-001 CD-ROM driver (IBMCDRT.DLL).



CD Audio Media Control Driver DLL (CDAUDIOT.DLL)

The CD audio MCD, CDAUDIOT, receives commands from the MDM and processes them. If a command message cannot be completely processed, it is sent to the hardware-specific code in the VSD.

Following describes the files and routines included in the CDAUDIOT.DLL.

CDAUDIO.C File

This module includes the entry point for the DLL. Global validation and parameter checking is performed on requested command messages. If the preliminary check finds no errors, a specific-processing function is called.

Procedure	Description
mciDriverEntry	Specifies the entry point to the MCD from the MDM.
pre_process_msg	Gets the device ready to process the command.
process_msg	Processes the requested command message.
verify_entry	Verifies that entry parameters are valid.
QMAudio	Queries the master audio's current settings.
Register	Registers the drive.
ReRegister	Registers the disc and tracks.
VSDReturn	Specifies the entry point to the MCD from the VSD. Processes return information from the VSD.
SetTrackInst	Sets track information in the instance.
ValPointer	Validates the address to record structures.

CDAUDPRO.C File

This module contains the hardware independent code that processes the command message. Some command may be processed completely by the MCD while others require the VSD to access the hardware or know of information specific to the VSD's associated drive.

Procedure	Description
ProcClose	Processes the MCI_CLOSE command.
ProcConnector	Processes the MCI_CONNECTOR command.
ProcCue	Processes the MCI_CUE command.
ProcCuePoint	Processes the MCI_SET_CUEPOINT command.
ProcGeneral	Processes pass through MCI commands.
ProcCaps	Processes the MCI_GETDEVCAPS command.
ProcInfo	Processes the MCI_INFO command.
ProcMAudio	Processes the MCI_MASTERAUDIO command.
ProcOpen	Processes the MCI_OPEN command.

ProcPause	Processes the MCI_PAUSE command.
ProcPlay	Processes the MCI_PLAY command.
ProcPosAdvise	Processes the MCI_SET_POSITION_ADVISE command.
ProcRestore	Processes the MCIDRV_RESTORE command.
ProcResume	Processes the MCI_RESUME command.
ProcSave	Processes the MCIDRV_SAVE command.
ProcSeek	Processes the MCI_SEEK command.
ProcSet	Processes the MCI_SET command.
ProcSetSync	Processes the MCI_SET_SYNC_OFFSET command.
ProcStatus	Processes the MCI_STATUS command.
ProcStop	Processes the MCI_STOP command.
ProcSync	Processes the MCIDRV_SYNC command.

CDAUDUTL.C File

This module contains the hardware independent code that supplement the process commands in CDAUDPRO.C. It also contains utility functions as well.

Procedure	Description
SetAudio	Sets audio information from MCI_SET.
SetConnector	Enables or disables a connection.
SetCuePoint	Enables the cue point.
StatusMCD	Gets status from MCD information.
StatusMCDDef	Gets status from MCD default information.
StatusVSD	Gets status from VSD information.
DisableEvents	Disables cuepoints and position advise.
GetTimeAddr	Converts a time format to/from MMTIME.
GetTimeAddrRC	Colinizes return code to time format.
GetTrackInfo	Gets the track information for a specified track.
ValAddress	Validates addresses to be in range.
ValState	Validates state of logical device.
vsdResponse	Processes VSD response.

CDMCINIT.C File

This module initializes the re-entrant DLL.

Procedure	Description
_DLL_InitTerm	Specifies the entry point for the OS/2 loader.
CDMCInitialization	Obtains initial heap.
CDMC_Exit	Cleans up instances after termination.

CDMCCOMN.C

This module contains the common functions between the CD MCD and the VSD for the IBM CD-ROM Drive.

Procedure	Description
parse_DevParm	Parses the device-specific parameter.
get_token	Gets next token and null terminates it.

HHPHEAP.C File

This module contains the common heap management functions.

Procedure	Description
HhpCreateHeap	Creates the first heap.
NewHeap	Creates a new heap (one segment).
HhpAllocMem	Allocates some memory.
LocateFreeMem	Locates a free block of memory.
HhpFreeMem	Frees memory.
CollapseFreeBlock	Removes fragmentation within a heap.
CollapseFreeHeap	Removes fragmentation within heaps.
HhpDestroyHeap	Destroys a heap.
HhpAllocBuffer	Allocates a buffer from the system.
HhpFreeBuffer	Frees the allocated buffer.
HhpAccessBuffer	Sets up segment for access.
HhpAccessHeap	Accesses a shared heap.
HhpReleaseHeap	Releases a heap.
AddPid	Adds a process ID to the PID list.
ReallocHeap	Reallocates a heap.
HhpGetPID	Gets the PID for the heap.
HhpDumpHeap	Dumps heap contents to standard out.

Vendor-Specific Driver for CD-ROM Drives (IBMCDRT.DLL)

Following describes the files and routines included in the IBMCDRT.DLL.

CDMCINIT.C File

See table in previous section.
CDMCCOMN.C File

See table in previous section.
HHPHEAP.C File

See table in previous section.
IBMCDROM.C

This module includes the entry point for the DLL. It contains the device dependent code for a specific CD-ROM drive.

Procedure	Description
vsdDriverEntry	Specifies the entry point to the VSD from the MCD.
process_msg	Processes the requested command message.
CDAudClose	Error recovery routine.
CDAudErrRecov	Closes an instance.
CDAudInfo	Returns information about the component.
CDAudOpen	Opens an instance.
CDAudRegDisc	Registers a disc for the logical device.
CDAudRegDrive	Registers a drive for the calling MCD.
CDAudSet	Sets various attributes of the device.
CDAudSetVerify	Tests flags for the SET command.
CDAudStatus	Returns the requested attribute.
CDAudStatCVol	Returns mapped component volume levels.

IBMCDPRO.C File

This module processes hardware requests.

Procedure	Description
CD01_Cue	Prerolls a drive.
CD01_CuePoint	Sets up a cue point.
CD01_GetCaps	Gets device capabilities.
CD01_GetDiscInfo	Gets status information of the disc.
CD01_GetID	Gets the CD ID from the disc.
CD01_GetPosition	Gets the position of the head.
CD01_GetState	Gets the state of the device.
CD01_GetTOC	Returns the table of contents

	(MMTOC form).
CD01_GetVolume	Gets the volume settings of the drive.
CD01_LockDoor	Locks or unlocks the drive door.
CD01_Open	Opens the specified device or drive.
CD01_Play	Initiates a play operation.
CD01_PlayCont	Continues a play operation.
CD01_PosAdvise	Sets up a position advise command.
CD01_RegTracks	Registers tracks on the disc.
CD01_Restore	Restores the saved instance.
CD01_Resume	Unpauses a CD play operation.
CD01_Save	Saves the current instance.
CD01_Seek	Seeks to a particular redbook address.
CD01_SetVolume	Sets the volume of the drive.
CD01_Stop	Stops a CD play operation.

IBMCDMSC.C File

This module processes miscellaneous functions such as the timer routine for the play command, identify the drive, and package the IOCtl calls.

Procedure	Description
CD01_StartPlay	Starts the play operation.
CD01_Sync	Sync to MDM request.
CD01_Timer	Timer routine for play operation.
CD01_Timernotify	Timer routine to setup/notify events.
GetTableName	Gets the CD table full path name.
OpenFirstTime	First time device open tests.
QueryTable	Queries the CD look-up table.
CallIOCtl	Calls the hardware by way of IOCtls.

Resource Units and Classes

Device contexts are managed by the MDM, using an abstract concept of resource units, resource classes and valid class combinations.

Resource units provide a measurement for the resource manager to determine how many device contexts may be active at any given time. Each device specifies how many resource units it can process concurrently.

Besides a total resource number, each resource class has a maximum number of resource units available to it. This allows the MDM to determine how many device contexts from a particular class can be active concurrently. During the install procedure the maximum numbers

are provided.

On the MCI_OPEN of a device context the required resource units and class for the device context is returned in the MMDRV_OPEN_PARMS structure. The required resource units and resource class of a device context can be changed by the MCD by calling the MDM with the MCIDRV_CHANGERESOURCE message. For example, if a waveaudio device allocated 1 resource unit for a mono wave and two units for a stereo wave, then a load command might change the required units for that device context.

The final piece of resource management is provided during install. This is the valid class combinations. A certain device might have multiple classes, but not allow certain classes to have active device contexts concurrently. The following example is the Pro AudioSpectrum 16 card. It has at most two resource units available. It uses 2 classes, one for waveform audio and the other for MIDI. Each class can have at most one resource unit consumed by a active device context. It can have any number of in-active device contexts. Finally both classes can have active device contexts concurrently. This says that this card can support one waveform audio and or one MIDI.

Inserting Pages in the Multimedia Setup Notebook

This section explains how to insert settings pages in the Multimedia Setup notebook-a program which provides a user interface to the properties of multimedia devices that are registered with the Media Device Manager (MDM).

There are two approaches to insert settings pages in the Multimedia Setup notebook. The following discussion focuses on the first approach, which illustrates how to use the MCI_DEVICESETTINGS message to insert a page for a particular MCD. The second approach is discussed in Defining Changes to Other INI Files in Installation Requirements. This approach addresses a registration mechanism for Multimedia Setup to insert settings pages based on device type (not MCD). For example, pages that apply to the system or to all media control interface devices of a particular class.

Note:

> The MMSYSTEM.H header file defines data structures used in inserting a page in the Multimedia Setup notebook for a device object.
>
> Refer to the *OS/2 PM Reference* for additional information on notebook control window processing.

You can insert a notebook page in the Multimedia Setup program if device-specific properties exist for a particular device. When the Multimedia Setup program is creating a notebook window, it checks the MCI_SYSINFO_DEVICESETTINGS style bit in the *ulDeviceFlag* field of the MCI_SYSINFO_LOGDEVICE data structure. This data structure (as shown in the following example contains information about a logical device that is installed in the system. The MCI_SYSINFO_DEVICESETTINGS style bit indicates that the MCD has custom device settings pages.

```
typedef struct _MCI_SYSINFO_LOGDEVICE {
  CHAR    szInstallName[MAX_DEVICE_NAME];      /* Device install name            */
  USHORT  usDeviceType;                        /* Device type number             */
  ULONG   ulDeviceFlag;                        /* Flag indicating whether device */
                                               /*  device is controllable or not */
  CHAR    szVersionNumber[MAX_VERSION_NUMBER]; /* INI file version number        */
  CHAR    szProductInfo[MAX_PRODINFO];         /* Textual product description    */
  CHAR    szMCDDriver[MAX_DEVICE_NAME];        /* MCI driver DLL name            */
  CHAR    szVSDDriver[MAX_DEVICE_NAME];        /* VSD DLL name                   */
  CHAR    szPDDName[MAX_PDD_NAME];             /* Device PDD name                */
  CHAR    szMCDTable[MAX_DEVICE_NAME];         /* Device-type command table      */
  CHAR    szVSDTable[MAX_DEVICE_NAME];         /* Device-specific command table  */
  USHORT  usShareType;                         /* Device sharing mode            */
  CHAR    szResourceName[MAX_DEVICE_NAME];     /* Resource name                  */
  USHORT  usResourceUnits;                     /* Total resource units available */
                                               /*  for this device               */
  USHORT  usResourceClasses;                   /* Number of resource classes for */
                                               /*  this device                   */
  USHORT  ausClassArray[MAX_CLASSES];          /* Maximum number of resource     */
                                               /*  units for each class          */
  USHORT
ausValidClassArray[MAX_CLASSES][MAX_CLASSES];/* Valid class combination */
} MCI_SYSINFO_LOGDEVICE;
```

If an MCD creates one or more custom settings pages, the Multimedia Setup program sends a MCI_DEVICESETTINGS message to the MCD. This allows the MCD the opportunity to insert device-specific pages in the Multimedia Setup notebook for a particular device. The MCD then passes this message to a Vendor-Specific Driver (VSD) if one exists. Some device properties might be related to the type of device and some might be related to the particular manufacturer's hardware.

The MCI_DEVICESETTINGS_PARMS data structure (as shown in the following example) defines the information that is passed to the MCD in the MCI_DEVICESETTINGS message. The *hwndNotebook* field contains the window handle of a CUA notebook control window where the page should be inserted. The *usDeviceType* field defines the type of media device, and the *pszDeviceName* field defines the logical device name (WAVEAUDIO01) of the device for which custom settings are to be inserted.

```
typedef struct_MCI_DEVICESETTINGS_PARMS {
ULONG     hwndCallback;             /* Window handle             */
HWND      hwndNotebook;             /* Handle to notebook window  */
USHORT    usDeviceType;             /* Device type               */
PSZ       pszDeviceName;            /* Device name               */
 } MCI_DEVICESETTINGS_PARMS;
```

You should also provide help for each page inserted in the notebook. The recommended way of implementing help for a Multimedia Setup settings page is to handle the WM_HELP message explicitly in the dialog procedure and then send the HM_DISPLAY_HELP message to a help instance that is created by the page (or group of pages).

When help for a tab is requested the Multimedia Setup program sends a MM_TABHELP message to the page as shown in the following example. The page window procedure can then display the appropriate help panel.

```
MM_TABHELP
   mp1          ULONG ulPageID       /* Page identifier           */
   mp2          ULONG Reserved
   return       TRUE for handled and FALSE not handled
```

Some pages use the device-specific parameters of the MCD to save settings information. New pages should expect that other pages are also using the device-specific parameters to save their settings and therefore other keywords may exist or exist in the future. New pages should be implemented in such as way as to preserve keyword values that they do not recognize.

The following example provides source code to create a modeless secondary window and insert a page in a notebook. This code can be provided as a part of an external DLL or part of the MCD code itself. The page should be consistent with the existing notebook design and be inserted after the standard pages in the notebook.

```
HWND InsertExamplePage( LPMCI_DEVICESETTINGS_PARMS pMCIDevSettings)
{
  HWND  hwndPage;               /* Page window handle                  */
  CHAR  szTabText[CCHMAXPATH];  /* Buffer for tab string               */
  ULONG ulPageId;               /* Page Identifier                     */

  /*********************************************************************/
  /*  Load a modeless secondary window.                                */
  /*********************************************************************/

  hwndPage    = WinLoadSecondaryWindow(
                      pMCIDevSettings->hwndNotebook,
                      pMCIDevSettings->hwndNotebook,
                      ExamplePageDlgProc,
                      vhmodMRI,
                      ID_EXAMPLE,
                      (PVOID)pMCIDevSettings);

  if (!hwndPage) return (NULL);

  ulPageId = (ULONG)WinSendMsg( pMCIDevSettings->hwndNotebook,
                      BKM_INSERTPAGE,
                      (ULONG)NULL,
                      MPFROM2SHORT( BKA_AUTOPAGESIZE | BKA_MINOR, BKA_LAST ) );

  /*********************************************************************/
  /*  Associate a secondary window with a notebook page.               */
  /*********************************************************************/

  WinSendMsg(pMCIDevSettings->hwndNotebook, BKM_SETPAGEWINDOWHWND,
            MPFROMP( ulPageId ), MPFROMLONG( hwndPage ) );

  /*********************************************************************/
  /* Get tab text from DLL.                                            */
  /*********************************************************************/

  WinLoadString(WinQueryAnchorBlock( HWND_DESKTOP ), vhmodMRI,
         (USHORT)IDB_EXAMPLE, CCHMAXPATH, szTabText );
```

```
  /*********************************************************************/
  /* Set tab text.                                                     */
  /*********************************************************************/

  WinSendMsg( pMCIDevSettings->hwndNotebook, BKM_SETTABTEXT,
        MPFROMP( ulPageId ), szTabText );

  return( hwndPage );

}

 typedef struct {
    HWND hwndHelpInstance;
 } MMPAGEINFO;
 typedef MMPAGEINFO * PMMPAGEINFO;

  /*********************************************************************/
  /* Modeless secondary window procedure                               */
  /*********************************************************************/

 MRESULT EXPENTRY ExamplePageDlgProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)

    PMMPAGEINFO pMMPageInfo = (PMMPAGEINFO) WinQueryWindowPtr (hwnd, QWL_USER);

    switch (msg) {
       case WM_INITDLG:
          /*****************************************************************/
          /*  Place window initialization code here.                       */
          /*****************************************************************/
          pMMPageInfo = (PMMPAGEINFO) malloc(sizeof(MMPAGEINFO));
          WinSetWindowPtr (hwnd, QWL_USER, pMMPageInfo);

          /*****************************************************************/
          /*  Create a help instance.                                      */
          /*****************************************************************/
          pMMPageInfo->hwndHelpInstance = WinCreateHelpInstance(...);
          break;

       case WM_DESTROY:
          /*****************************************************************/
          /*  Clean up page window resources.                              */
          /*****************************************************************/
          WinDestroyHelpInstance (pMMPageInfo->hwndHelpInstance);
          free (pMMPageInfo);
          break;
       case WM_COMMAND:
          /*****************************************************************/
          /*  Process all commands.                                        */
          /*****************************************************************/
          return ((MRESULT) FALSE);
          break;

       case MM_TABHELP:
          /*****************************************************************/
          /*  Display help for a tab.                                      */
          /*****************************************************************/
          if (pMMPageInfo->hwndHelpInstance) {
              WinSendMsg(
                    pMMPageInfo->hwndHelpInstance,
                    HM_DISPLAY_HELP,
                    MPFROMSHORT( WinQueryWindowUShort( hwnd, QWS_ID ) ),
                    HM_RESOURCEID );
          }
          break;

       case WM_CLOSE:
          return ((MRESULT) FALSE);
          break;

       case WM_HELP:
          if (pMMPageInfo->hwndHelpInstance) {
              WinSendMsg(
                    pMMPageInfo->hwndHelpInstance,
                    HM_DISPLAY_HELP,
                    (MPARAM) mp1,
                    HM_RESOURCEID );
           }
           return ((MRESULT)TRUE);
           break;
```

```
   case WM_TRANSLATEACCEL:
      return (WinDefWindowProc (hwnd, msg, mp1, mp2));
      break;

 case HM_QUERY_KEYS_HELP:
      return((MRESULT) IDH_HELPFORKEYS);
      break;

}

return (WinDefSecondaryWindowProc(hwnd, msg, mp1, mp2)); }
```

Stream Handlers

This section describes the Stream Programming Interface (SPI) services used by MCDs to implement data streaming and synchronization, as well as the interfaces used by stream handlers to transport data.

Stream Handler Architecture

Pairs of stream handlers implement the transport of data from a source to a target device while the Synchronization and Streaming Manager (SSM) provides coordination and central management of data buffers and synchronization data.

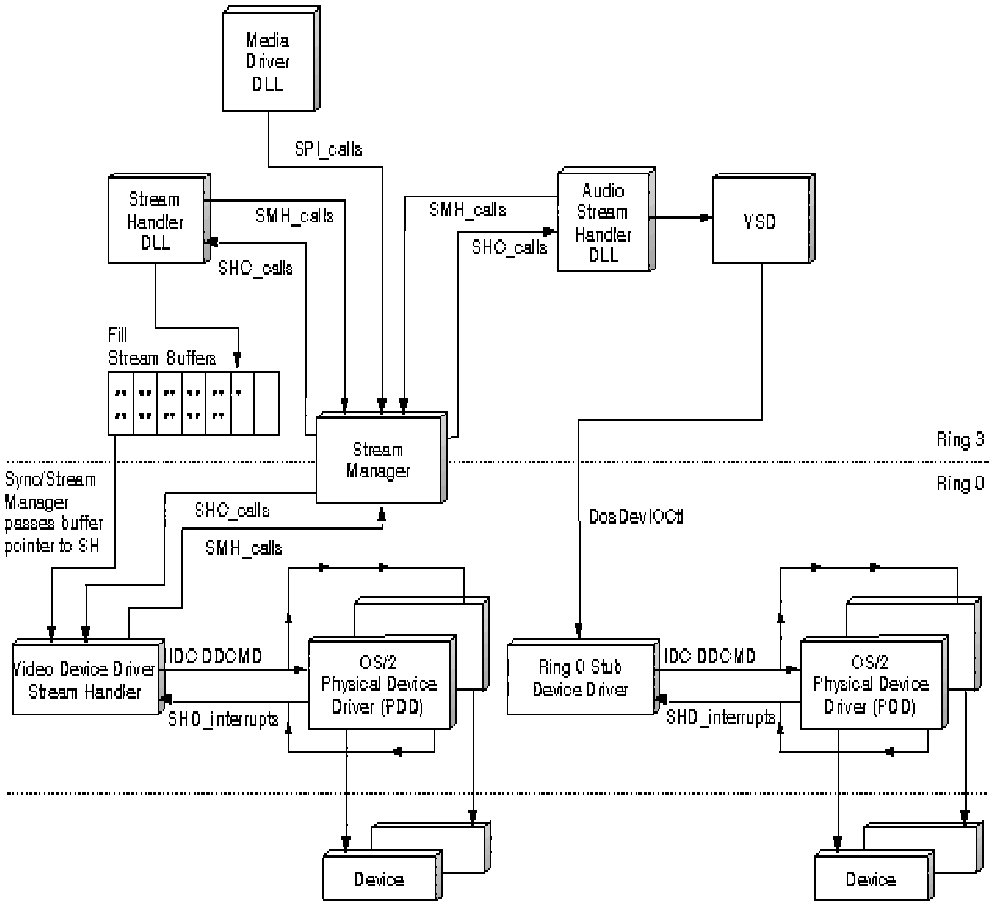
A stream handler can be built as a device driver run at Ring 0, or as a dynamic-link library (DLL) run at Ring 3. See DLL Model: File System Stream Handler and Device Driver Model: Video PDD for examples of these stream handler types.

Some streams are ideally controlled by a direct connection between the stream handler and a physical device driver, whereas other streams are not associated with a data source or target that maps physically to a specific device. For example, the File System Stream Handler is a DLL because all file system I/O functions are available as Ring 3 OS/2 functions that service all file system devices.

After you determine the type of stream handler you require (DLL or DD), you must incorporate the required functional modules in your stream handler design. Refer to the following figure, which represents the key modules in both DLL and device driver stream handler components.

Although the detailed coding practices of building OS/2 DLLs and device drivers differ considerably, most logic for a stream handler is not affected by this choice. The structure of the DLL stream handler closely resembles the device driver stream handler, with few exceptions.

The following figure illustrates the logical structure of the Sync/Stream Manager (SSM) and its relationship with stream handlers. The Sync/Stream Manager DLL exports SPI services to higher-level OS/2 multimedia components (such as media drivers) and exports Stream Manager Helper (SMH) messages to support the stream handler DLLs. Additional SMH messages are exported by the Sync/Stream Manager device driver to stream handler device drivers using standard OS/2 inter-device driver communication (IDC) interfaces (established using DevHelp_AttachDD).



The following table illustrates the stream handlers are provided by OS/2 multimedia.

Note: See Stream Handler Module Definitions for information describing the high-level design and operation of the stream handlers provided with OS/2 multimedia.

Stream Handler	Description
Audio	VSD interface to a vendor-specific driver. Supports PCM, MIDI, ADPCM formats.
MIDI Mapper	Filters data using the selected MIDI map.
File system	Uses file system services to read or write data from any associated device.
Multi-track	Reads and splits interleaved data.
Video	In conjunction with CODECs, outputs video data to the display.
System Memory	Transfers data to and from system memory buffers. See Cuepoint Event Support.
CD-ROM XA	Reads CD-ROM XA data and splits audio sectors from video and data sectors. See CD-ROM XA Stream Handler.
CD-DA	Reads digital audio data directly from the CD-ROM driver. Can be used to play back CD audio data using an audio adapter.

Synchronization Features

The ability to synchronize events is essential for multimedia applications. For example, you might decide to display a certain bitmap (an image of fireworks bursting) at precisely the same time you play an audio waveform (a specific cymbal crash during the "Star Spangled Banner"). From a standard OS/2 application perspective, two independent threads or a single thread might control these events. However, there is no way for either of these approaches to guarantee that both events will occur within a specified time span. The longer the time delay between events, the more likely a user will notice the loss of synchronization.

The sync/stream subsystem design supports multimedia application synchronization requirements by ensuring that program events such as the output of specific data elements (digital audio, image, MIDI audio, and so on) can be reliably synchronized within very narrow real-time limits. The sync/stream subsystem also reduces the complexity of application code required to synchronize events.

Being able to synchronize user-oriented and system-driven events is also important to multimedia application developers. For example, an application might need to adjust the tempo of a musical playback sequence while responding to the user's movement of the mouse or pressing arrow keys (゜,) on the keyboard. Any perceivable delay in response might be unacceptable. Of more urgent importance is a pilot training program that requires the trainee to press a key to respond to a ground-proximity alarm. If the trainee senses a delay between pressing the key and the alarm sound muting, it would be reasonable to expect the real cockpit controls to react the same way. This false expectation might lead to a crash in real life.

SSM synchronization functions simplify a programmer's work. For example, instead of establishing large and complex control structures to synchronize a group of streams, a programmer needs only to identify the streams as part of a group.

The sync/stream subsystem design includes several features that provide for simple, effective control of real-time event synchronization:

- Master/slave relationship
- Sync pulse generation
- Sync pulse processing
- Sync/stream subsystem events
- Null stream handler

These features exploit the multitasking capability of OS/2 to prevent specific regions of code and data in physical memory from being paged to disk. This ensures that the required modules can process data and requests in a real-time manner, avoiding the latency that would arise should those critical regions be swapped out when needed.

Master/Slave Relationship

A master/slave relationship is a specification of a chain of command for controlling a synchronized event. The relationship is 1:N, where one object (the synchronization *master*) controls the behavior of one or more subordinate objects (the *slaves*). The relationship is established using the SpiEnableSync function, where one data stream is designated the master and one or more data streams are designated as slaves. Real-time information, transmitted from the master to all slaves by the Sync/Stream Manager, gives each of the slaves the current time in the time base of the MMTIME standard (1/3 msec units). This time information (*sync pulse*) allows each slave stream handler to adjust the activity of that stream so that synchronization can be maintained.

Certain rules govern master/slave synchronization relationships:

- A data stream can be a slave in only one synchronization relationship.
- Stream handlers must be able to enslave multiple data streams (if multiple data streams are supported) according to different sync masters. For example, each active stream under that handler's control might have a different master.
- A stream cannot become the master of two separate groups of slaves (any additional streams become slaves in the existing master/slave group).
- The sync group (master and all slaves) can be started, stopped, and seeked as a group by using the "slaves" flag on each of the following SPI functions:
    - SpiStartStream
    - SpiStopStream
    - SpiSeekStream

- Any slave in a sync relationship may be arbitrarily started or stopped without affecting the activity of the master or any other slave. The master can be stopped independently of any slave streams in the same sync group and the option is available for all of the slave to be stopped at the same time that the master stream is stopped.
- Group synchronized stream time will always be the master stream time. This includes slave streams that have been stopped and re-started.
- Stream seeking is done on a per stream basis. For example, a seek in a master stream does not seek in any slave streams that may be in the same sync group.

It is possible for a slave to fail to maintain synchronization. This condition, called *sync overrun* , happens when a stream handler has not processed the last sync pulse received from the Sync/Stream Manager and another sync pulse is received. The application can optionally request (by way of the SpiEnableEvent function) to be notified of any sync overruns. The stream is not stopped but the application could request a stop once it receives a sync overrun event.

Sync Pulse Generation

A *sync pulse* represents the current stream sync master *clock value* and its stream handle (HSTREAM). The *clock value* is expressed in MMTIME units (1/3 milliseconds) where 0 represents the time the master stream was started from the beginning of its associated multimedia data object. If a seek operation is performed on the master, the stream must be stopped. The master stream time also stops when a seek takes place and is reset to the stream time of the seek point. When the stream is restarted, the stream time is restarted from the seek point. This means that stream time can be equated to position in the data as opposed to the amount of time the stream has been active.

Sync pulses are generated on a regular basis by the master. A slave receives sync pulses only when the Sync/Stream Manager determines that it is out of sync with the master. Sync pulses are distributed by the stream manager based on the programmed stream sync relationship. This distribution is effective for both DLL and device driver slave stream handlers.

Each slave stream handler must regularly update the sync pulse SYNC_EVCB with what it believes the stream time is. The Sync/Stream Manager checks this slave handler stream time against the master stream time and decides whether to send a sync pulse to this handler.

Device driver stream handlers receive sync pulses through their sync pulse event control block (SYNC_EVCB). A device driver stream handler must check for sync pulses from the Sync/Stream Manager by polling a flag in the sync pulse SYNC_EVCB. The Sync/Stream Manager sets the flag to indicate a sync pulse and updates the current master stream time. Usually, the device driver slave handler polls the flag once during interrupt processing and adjusts the stream consumption accordingly.

DLL stream handlers receive sync pulses in one of two ways. A DLL stream handler can register a semaphore with the Sync/Stream Manager, or it can use the same method the SSM uses for a device driver stream handler.

Sync Pulse Processing

Each stream handler (DLL or device driver) can provide sync pulse handling to support synchronization for any streams activated. Typically, the stream handler sync pulse handling logic is capable of correcting the active stream's progress in its real-time context, based on information received in sync pulse SYNC_EVCB created by the stream handler and accessed by the Sync/Stream Manager.

For example, if the slave's current local time is 45000 (15.0 seconds in MMTIME), and the master sync pulse time is 44500 (14.83 sec), then the slave stream handler's sync pulse logic should adjust the pace of data streaming (slow down, or repeat the last 170 milliseconds worth of data). This time-adjusting (resync) logic is unique to the type of data the handler is streaming.

Since there can be occurrences of heavy system load that make it impossible for DLL sync pulse logic to get control between two sync pulses (task dispatch latency can rise occasionally to delay even time-critical threads), the handler always takes the most current master time (last sync pulse posted). The SSM sets a sync pulse *overrun flag* if a sync pulse is not processed before the next sync pulse arrives. Also, the sync pulse logic and all associated control data objects must reside in fixed memory, so that these code and data pages are never paged out (at least not while the stream is active).

The sync pulse logic must be high performance by nature. Complex, time-consuming synchronization adjusting routines with very long path lengths will not be able to maintain synchronization if they are dispatched continuously. The sync pulse logic is a second level interrupt handler, so its performance must be optimized. If no adjustment of the stream is necessary, the routine might not need to be dispatched at all, which brings us to one final point.

The Sync/Stream Manager has sync pulse distribution logic that controls when it will set the slave device driver handler's sync pulse SYNCPOLLING bit flag (the SSM does not call the handler's IDC interface). This logic similarly controls when it will dispatch the DLL handler's time-critical thread (clear a semaphore the thread is waiting on). Typically, the DLL handler detects that a sync pulse has occurred

by polling the SYNCPOLLING bit flag in its sync pulse SYNC_EVCB that it gave to the Sync/Stream Manager on return from an SHC_ENABLE_SYNC message. This SYNCPOLLING bit flag is set only when the difference between the master stream time and the slave's local time (both stored in the sync pulse SYNC_EVCB) exceeds a specified *resync tolerance value* . The Sync/Stream Manager determines this difference for all slaves on each sync pulse received from the master, and only notifies the slaves whose stream has deviated too much from the master stream time. Each stream handler that can receive sync pulses must provide its tolerance value (a minimum time difference expressed in MMTIME) in the stream's SPCB passed on return from the SHC_CREATE.

Sync/Stream Subsystem Events

Sync/stream subsystem events identify a specific change of state for either a master or slave stream handler. Some of these events do not have any effect on synchronization and merely indicate status of the data stream.

Two classes of events are defined: implicit and explicit. *Implicit* events are those events which all stream handlers must always support (such as end of stream or preroll complete). *Explicit* events are events which are supported only by some handlers (such as a custom event unique to a particular type of data). The application automatically receives notification of implicit events; however, the application must enable explicit events by using the SpiEnableEvent function if it wants to receive notification of any other events. Events remain enabled until they are disabled. The SSM and stream handlers are the only components that generate events.

The following table lists the events that are currently defined for the sync/stream subsystem. Stream handlers can also define new events for use as a communication mechanism between the stream handler and the application. Refer to the *ulType* and *ulSubType* fields in the EVCB data structure in the *OS/2 Multimedia Programming Reference* .

Event	Data Structure	Description
EVENT_CUE_DATA (Explicit)	DATA_EVCB	A cuepoint in terms of some specific piece of data in a stream. This event can be enabled as a single event or as a recurring event.
EVENT_CUE_TIME (Explicit)	TIME_EVCB	A cuepoint in terms of stream time from the start of the stream. This event can be enabled as a single event or as a recurring event. Recurring events are events that are defined as a time interval. An event is generated an each occurrence of this time interval. Single events remain enabled, even after they are reported. In case the stream is seeked backwards in time to a position before a cuepoint and play is resumed, the cuepoint will be reported again.
EVENT_CUE_TIME_PAUSE (Explicit)	TIME_EVCB	A cuepoint in terms of stream time from the start of the stream. This event will cause the stream to be paused when the cuepoint is reached. This event can be enabled as a single event.
EVENT_DATAOVERRUN (Explicit)	EVCB	A stream handler detected a data overrun. Data could be lost in a recording situation.
EVENT_DATAUNDERRUN (Explicit)	EVCB	A target stream handler detected a data underrun condition. There was no data available to output to the output device. Usually in this situation, the target stream handler will attempt to get another buffer and then pause it's device. The target stream handler will be re-started when more data is available to be output. This condition results in a break in the output data stream. Interleaved data format can cause underruns to occur when the end of the data is reached, but the end of file has not been reached.
EVENT_EOS (Implicit)	IMPL_EVCB	End of stream event. This event is generated after the target stream

		handler has consumed the last buffer in the stream. This signals to the application that the stream has completed processing.
EVENT_ERROR (Implicit)	IMPL_EVCB	An error has occurred while streaming.
EVENT_PLAYLISTCUEPOINT (Implicit)	PLAYL_EVCB	A memory stream handler playlist cuepoint event.
EVENT_PLAYLISTMESSAGE (Implicit)	PLAYL_EVCB	A memory stream handler playlist message.
EVENT_QUEUE_OVERFLOW (Implicit)	IMPL_EVCB	Event queue overflow. Indicates that an event has been lost due to too many events being generated. The application (MCD) should use this event to clear any waiting conditions.
EVENT_STREAM_STOPPED (Implicit)	IMPL_EVCB	The stream has been emptied or discarded. (See the SpiStopStream function in the *OS/2 Multimedia Programming Reference*.)
EVENT_SYNC_PREROLLED (Implicit)	IMPL_EVCB	All synchronized streams are prerolled. (See the SpiStartStream function in the *OS/2 Multimedia Programming Reference*.)
EVENT_SYNCOVERRUN (Explicit)	OVRU_EVCB	A sync overrun has been detected in the stream.

The Sync/Stream Manager propagates events to the application through the media driver by doing a call-back to an event routine that the application registered with the SSM on an SpiCreateStream call. Use the event routine as if it were a second-level interrupt routine and do not attempt to do a lot of processing. The event routine is on a per-process basis; it receives only one process at a time. Therefore, when an event is sent to be processed, it must wait until the current event has completed processing.

Because there is only one EVCB for implicit events, it is a good idea to copy any needed information from the EVCB into local variables for processing by the event routine. The event routine has the following interface:

```
Event_Entry(PEVCB pEVCB, rc)
```

Null Stream Handler

The SSM provides a *null stream* creation capability to enable multimedia applications to synchronize nonstreaming devices with streaming devices. The null stream is created and started for the nonstreaming device, but has no data flow associated with it. The Duet Player II Sample Program provides an example of synchronizing the CD-DA device, which is a nonstreaming device.

Any application or media driver can create a null stream, or several null streams. Since the null stream has no default resync tolerance value, the creating module must establish this value by calling SpiGetProtocol followed by SpiInstallProtocol, modifying the resync tolerance field between these two calls. This establishes the frequency of event notification to the calling process, allowing the system to maintain sync without the burden of running the notification thread on every sync pulse.

A given process can create multiple null streams. These streams are notified of sync pulses in the same way as real stream handlers; therefore, each null stream handler must support some capabilities a real stream handler supports. Each null stream thus created can have different sync pulse timing characteristics. Sync pulse granularity for each stream is governed by changing the resync tolerance value in the respective SPCBs.

Null stream handlers must support the following Stream Handler Command (SHC) messages:

- SHC_CREATE
- SHC_DESTROY
- SHC_DISABLE_EVENT
- SHC_DISABLE_SYNC
- SHC_ENABLE_EVENT
- SHC_ENABLE_SYNC

- SHC_ENUMERATE_PROTOCOLS
- SHC_GET_PROTOCOL
- SHC_GET_TIME
- SHC_INSTALL_PROTOCOL
- SHC_NEGOTIATE_RESULT
- SHC_SEEK
- SHC_START
- SHC_STOP

Stream Protocols

The Stream Protocol Control Block (SPCB) defines key parameters that control the behavior of a data stream. The application can query, install, or deinstall a specific SPCB from a stream handler. SPCB information is installed in the SPI.INI file by means of a resource file. (See Installing a Stream Handler for a description of how to install stream protocol.)

Each stream handler supports one or more stream protocols. Each SPCB is uniquely identified by the value of the stream data type (SPCB key). One field in the SPCB key allows the stream handler to have multiple SPCBs installed for the same data type. This field can be used by an application to specify which SPCB, for any data type, it wants to use. Each application in the system could define multiple SPCBs for the same data type (see the *ulType* field in the SPCBKEY data structure). The application can modify a stream protocol by installing a new SPCB and deinstalling the old SPCB.

The parameters in the SPCB are:

Fields	**Meaning**
SPCB Length	Length of SPCB structure. Provided for future expansion of the SPCB.
SPCB Key	Data stream type and internal key. The internal key is used to differentiate between multiple SPCB's of the same data stream type. • Data Type - The data type of the stream (for example, waveform or MIDI). • Data Subtype - The subtype of the stream (for example, 16-bit stereo PCM). • Internal Key - Used to differentiate between multiple SPCB's of the same data type and subtype.
Data Flag	Attributes of the data type. (that is, specifies whether data or time cuepoints are supported by this data type). • SPCBDATA_CUETIME - This data type can support time cuepoint events. • SPCBDATA_CUEDATA - This data type can support data cuepoint events. • SPCBDATA_NOSEEK - Seeking cannot be done on this data type.
Number of Records	Maximum number of records per buffer. (This is only valid for split streams).
Data Block Size	Size of a block. A block is an atomic piece of data. For example, for digital audio data type PCM 16-bit stereo at a 44.1 kHz sampling rate, the block size would be 4 bytes.
Data Buffer Size	Size of buffer to be used while streaming. Maximum buffer size is 64KB.
Min Number of Buffers	Minimum number of buffers needed to maintain a constant data stream.
Max Number of Buffers	Maximum number of buffers needed (ideal number). For normal streams, this means the number of buffers that will be allocated for the stream. For user provided buffer streaming, this means the number of buffers that the Sync/Stream Manager can queue up for a consumer. This can be used by a source stream handler that gives the same set of buffers to the Sync/Stream Manager repeatedly. If the number of buffers is set to the number of buffers in the set minus one, the source stream handler will be able to detect when the target has consumed a buffer and it can be reused. Assuming that the set of buffers is an ordered set and each buffer is used in the same order each time.

Source Start Number	Number of EMPTY buffers required to start the source stream handler. The value should be at least as big as the maximum number of buffers that would be requested by the source stream handler.
Target Start Number	Number of FULL buffers required to start the target stream handler. The value should be at least as big as the maximum number of buffers that would be requested by the target stream handler. Usually, a target will require at least two buffers at the start of the stream.
Buffer Flag	Buffer attributes (that is, the user provides buffers, fixed block size, interleaved data type, and maximum buffer size). • SPCBBUF_USERPROVIDED - User provides buffers for streaming. The Sync/Stream Manager will not allocate buffers, but attempt to lock down user provided buffers or copy the data to locked buffers. Using this flag will affect the performance of streaming. Only a source stream handler can set this flag. This flag is mutually exclusive with the SPCBBUF_INTERLEAVED flag. The SPCBBUF_FIXEDBUF cannot be used with this flag set. • SPCBBUF_FIXEDBUF - The buffer size for this stream handler must be a particular fixed size (for this data type). This flag cannot be used with the SPCBBUF_USERPROVIDED flag. The SPCBBUF_INTERLEAVED flag (split stream) implies SPCBBUF_FIXEDBUF. • SPCBBUF_NONCONTIGUOUS - Each data buffer is allocated contiguously in physical memory unless both stream handlers set the non-contiguous flag (SPCBBUF_NONCONTIGUOUS). This flag allows the system flexibility in allocating memory. A device driver stream handler may require contiguous memory, while a DLL stream handler may not. • SPCBBUF_INTERLEAVED - This stream is a split stream. It consists of one input stream of interleaved data that is split into multiple streams of individual data types. Only the source stream handler can set this flag. This flag is mutually exclusive with the SPCBBUF_USERPROVIDED flag. The SPCBBUF_FIXEDBUF cannot be used with this flag set. • SPCBBUF_MAXSIZE - The *ulBufSize* field contains that maximum size buffer that this stream handler can handle. • SPCBBUF_16MEG - The stream buffers may be allocated above 16MB. This is used by stream handlers device drivers that can support greater than 16MB addresses.
Handler Flag	Stream handler flags (that is, the handler can generate and receive sync pulses, and use the Sync/Stream Manager timer as the master, non-streaming handler). • SPCBHAND_GENSYNC - This stream handler can generate sync pulses. • SPCBHAND_RCVSYNC - This stream handler can receive sync pulses. • SPCBHAND_NONSTREAM - This stream handler is a non-streaming handler (that is, a stream handler that can participate in synchronization, but does not stream). • SPCBHAND_GENTIME - This stream handler can keep track of the real stream time. This handler also supports the *SpiGetTime* and cuepoint events. • SPCBHAND_NOPREROLL - This stream handler cannot preroll its device (that is, for recording streams, the source stream handler cannot be prerolled). The Sync/Stream Manager will treat this stream as if it were prerolled if asked to preroll this stream. • SPCBHAND_NOSYNC - This stream handler can be in a sync group but does not receive or generate sync pulses. (It is useful to group streams together so that they can be manipulated as a group). • SPCBHAND_PHYS_SEEK - This stream handler does a seek to a physical device. Other stream handlers only adjust stream time on a seek request. The Sync/Stream Manager will always call this stream handler first on a call to the *SpiSeekStream* function.
Resync Tolerance	Resync tolerance value. It is used to determine whether to send a sync pulse to this specific slave stream handler, if it is a slave.
Sync Pulse Granularity	Time interval in MMTIME units between sync pulses. Used to save sync pulse generation granularity if this stream handler is a master, but cannot generate its own sync pulse.

Bytes Per Unit	Bytes per unit of time. This is used to do seeks on linear data that is not compressed or of variable length. It is also used for SHC_GETTIME queries in a stream handler.
MMTIME Per Unit	The amount of MMTIME each unit represents. This is also used for the seek and "get time" functions of a stream handler.

Refer to the *OS/2 Multimedia Programming Reference* for further information on structure and flag definitions.

Stream Protocol Negotiation at Stream Creation Time

Following are some statements regarding stream protocol negotiation.

- The following flags are set to On in the negotiated SPCB for the stream handler that should perform the action specified by the flag. Either the source or the target negotiated SPCB will contain these bit flags.

SPCBHAND_GENSYNC	SPCBHAND_RCVSYNC
SPCBHAND_GENTIME	SPCBHAND_PHYS_SEEK

- The DATATYPE_GENERIC will match any other data type during negotiation. This is useful for stream handlers like the File System Stream Handler that can be a source for almost any data type.
- If neither of the SPCB key data types are generic, then both the data type and subtype fields must match or an error will occur. The internal key field is not used during negotiation. The internal key of 0 is returned from negotiation.
- The block size will default to 1 byte if not specified. The source and target block size must match or the negotiation will fail.
- The Data Buffer Size must be a multiple of block size.
- The negotiation will fail if one stream handler has a fixed buffer size (SPCBBUF_FIXEDBUF) greater than the maximum buffer size (SPCBBUF_MAXSIZE) of the other.
- Both handlers must not have fixed buffer sizes (SPCBBUF_FIXEDBUF) of different lengths.
- Both handlers must not have maximum buffer sizes (SPCBBUF_MAXSIZE) of different lengths.
- Negotiation will default to a fixed buffer size (SPCBBUF_FIXEDBUF). Otherwise, the buffer size is set the greater of the two SPCB buffer sizes, but not greater than the maximum buffer size (SPCBBUF_MAXSIZE), if one is specified.
- If no special conditions (SPCBBUF_FIXEDBUF,SPCBBUF_MAXSIZE,SPCBBUF_USERPROVIDED) were specified, the largest buffer size is the one that will be used for the stream creation.
- For user-provided buffers (SPCBBUF_USERPROVIDED), the buffer size is set to the maximum buffer size (SPCBBUF_MAXSIZE), or to the largest buffers possible rounded to a multiple of the block size.
- Split streams (SPCBBUF_INTERLEAVED) must have a maximum number of records per buffer greater than 0. The target stream handler must set this field to 0, since only the source stream handler can set this value.
- The Minimum and Maximum number of buffers must be greater than zero.
- The largest Minimum number of buffers value is used.
- The largest Maximum number of buffers value is used.
- The Sync/Stream Manager attempts to allocate the maximum number of stream buffers requested. If it is unable to allocate this amount of space, but is able to allocate the minimum needed, the stream is created. Otherwise, the stream creation fails for lack of resources.
- The number of EMPTY buffers required to start the source is always taken from the source SPCB.
- The number of FULL buffers required to start the target is always taken from the target SPCB.
- For SpiGetTime requests, each handler must specify whether it can receive these requests and return real-time information. It can do this by specifying the SPCBHAND_GENTIME flag. For negotiation, the target stream handler is the default provider of this information unless only the source can provide this information.

- The Bytes Per Unit and MMTIME Per Unit are set from the stream handler that handles the real-time requests per the previous statement.
- For sync pulses, each handler must specify whether it can send or receive sync pulses. It can do this by specifying the SPCBHAND_GENSYNC or SPCBHAND_RCVSYNC flag. For negotiation, the target stream handler is the default generator and receiver of sync pulses unless only the source can generate and receive sync pulses.
- The sync tolerance is only valid for handlers that set the SPCBHAND_RCVSYNC.
- The sync pulse granularity is only valid for handlers that set the SPCBHAND_GENSYNC.
- The SPCBHAND_PHYS_SEEK flag is used to specify if a stream handler does a physical device seek when called on an SHC_SEEK call. The Sync/Stream Manager uses this information to determine which stream handler should be called first on an *SpiSeekStream* call. The handler that does a physical device seek will be called first, otherwise, the stream handler that specified SPCBHAND_GENTIME will be called last.
- Any reserved fields must be set to NULL.
- Any undefined bit in any bit flag of the stream protocol must be set to 0.

Cuepoint Event Support

In certain instances it is not appropriate for a source stream handler to report cuepoints to an application (or media driver), but to allow the target to report the cuepoint. Usually, an application awaiting a cuepoint wants to receive it at the time the data is being output on the target side of a stream. One such circumstance involves the System Memory Stream Handler.

The System Memory Stream Handler has the ability to stream data from a *playlist.* A *playlist* is basically a little program that specifies what memory objects from a user application should be streamed and in what order. It consists of control information to enable looping, branching, and subroutines calls within the playlist. Another feature of a playlist is the CUEPOINT instruction, which allows the playlist to specify where in the data stream that cuepoints need to be sent to the application. Since the playlist is "played" within the source stream handler, it needs to notify the target stream handler to report the cuepoint at the appropriate time in the data stream.

The Sync/Stream Manager provides the ability for a source stream handler to associate a cuepoint with a buffer that it passes to the target stream handler (through the Sync/Stream Manager). This buffer and attribute remain in the data stream until the target stream handler requests it. The attribute for the buffer is passed with the buffer to the target (refer to the SMH_NOTIFY message in the *OS/2 Multimedia Programming Reference*). The target stream handler must then flag the event (for example, EVENT_CUE_DATA) when it receives a buffer with a non-zero attribute. The value of the attribute is passed as a parameter to the event on the SMH_REPORTEVENT call to the Sync/Stream Manager.

This feature of the Sync/Stream Manager supports the memory playlist ability to send a cuepoint event as close to actual time it occurs as possible. For this to work, it must be the target stream handler that must flag the cuepoint event to the application, in this case the media driver. If the System Memory Stream Handler is the source stream handler, it must notify the target stream handler to flag a cuepoint event. Refer to the *OS/2 Multimedia Programming Reference* for more information on playlists.

CD-ROM XA Stream Handler

The Sync/Stream Manager provides a mechanism to allow one data stream source and multiple targets, called *split streams* . For example, CD-ROM XA can interleave waveform audio, video, and text data within the same data buffer. The CD-ROM XA Stream Handler splits the data into multiple streams for CD-ROM XA devices. It is more efficient for the stream handler to read the combined data into one buffer and then insert the video data into a video stream and the waveform audio data into a waveform audio stream without copying the data. The waveform audio data stream is then consumed by a waveform audio stream handler and the video stream by a video stream handler. The waveform audio stream handler sees only the audio portions of the data stream and the video stream handler sees only the video portions of the stream.

A data stream can be split into multiple split streams. The number is limited only by the resources of the system. Each separate split of the stream is a separate stream. The work of parsing and determining the different data is the job of the source stream handler, which actually fills an empty buffer from the Sync/Stream Manager with data from the source device.

The source stream handler must determine which data goes in which stream. It must then return to the Sync/Stream Manager pointers to the individual records of the buffer. Each record is a portion of the buffer filled with data specific to one of the split streams. These records

are returned to the Sync/Stream Manager indicating which stream they should go to. The Sync/Stream Manager will then queue these records in the respective buffer or record queue for one of the split streams sharing the buffer.

You can set up this type of split stream as follows. Multiple streams must be created, each with a call to the *SpiCreateStream* function to the Sync/Stream Manager. The first of these functions is a normal stream creation call with the *hstreamBuf* parameter set to NULL. Every other subsequent *SpiCreateStream* call that needs to share the buffers of the first stream must pass that stream's handle in the *hstreamBuf* parameter. This is the mechanism used by the Sync/Stream Manager to allow these split streams to share the buffers of the first stream.

The source stream handler will also receive this parameter on an *SHC_CREATE* call and need to support this kind of streaming. The SPCBBUF_INTERLEAVED flag in the SPCB that is set by the source stream handler to indicate whether it can do split streaming. If it cannot, any *SpiCreateStream* attempting to use its buffers will be rejected. Any stream sharing buffers will be negotiated to use the same size buffers and number of buffers as the stream that allocated the buffers.

The split-stream mechanism only works for one source and multiple targets.

Streaming Scenarios

The following programming scenarios provide sample pseudocode of data streaming in the OS/2 multimedia environment using the sync/stream subsystem functions. This pseudocode shows how to structure your application code to use the SPI functions. For further details about how the sync/stream subsystem interfaces operate, refer to detailed definitions of the functions and messages in the *OS/2 Multimedia Programming Reference* .

Streaming Waveform Audio Data from the File System

This scenario describes creating a waveform audio data stream and associating a file containing waveform audio with the source stream handler (File System Stream Handler DLL), where the target handler is the Audio Stream Handler, as shown in the following example.

```
/************************************************************/
/*                        M A I N                           */
/************************************************************/
#include        <os2.h>
#include        <stdlib.h>
#include        <stdio.h>
#include        <conio.h>
#include        <string.h>
#include        "os2me.h"

main()
{
RC           ulRC;             /* Error return code */
SPCBKEY      SPCBkey;          /* Data type to stream */
IMPL_EVCB    EVCB;             /* Event control block for implicit events */
ACB_MMIO     acb;              /* Associate control block used to assoc */
                               /* The file to stream (play) */
DCB_AUDIOSH  dcb;              /* Audio device driver information */
HAND         ahand[5];         /* Enumerate handler info */
ULONG        ulNumHand = 5;
HID          hidSource, hidTarget, hidunused; /* Handler ID */
HSTREAM      hstream;          /* Stream handle */
HEVENT       hevent;           /* Event handle for implicit events */
HMMIO        hmmioIn;          /* Handle to mmio file open */
/* Get list of all stream handlers in system */
if (rc = SpiEnumerateHandlers(&ahand, &ulNumHand))
   return(rc);    /* error! */
/* Get the stream handler ID for the File System Stream Handler */
if (rc = SpiGetHandler("FSSH",&hidSource,&hidunused))
   return(rc);    /* error! */
/* Get the stream handler ID for the Audio Stream Handler */
if (rc = SpiGetHandler("AUDIOSH$",&hidunused,&hidTarget))
   return(rc);    /* error! */
/* Create a data stream from the file system to the */
/* waveform audio ACPA device connected to speakers. */
SPCBkey.ulDataType = DATATYPE_WAVEFORM;
SPCBkey.ulDataSubType = WAVE_FORMAT_4S16;
```

```
SPCBkey.ulINtKey = 0;
strcpy(dcb.szDevName,"AUDIO1$");
dcb.ulDCBLen = sizeof(dcb);
dcb.hSysFileNum = (** hSysFileNum returned by audio dd on audio_init call)
if (rc = SpiCreateStream(hidSource,
                         hidTarget,
                         &SPCBkey,
                         NULL,
                         (PDCB)&dcb,
                         (PEVCB)&EVCB,
                         EventProc,
                         0,
                         &hstream,
                         &hevent))
   return(rc);    /* error! */
/* USE MMIO to access a waveform audio object */
if ((hmmioIn = mmioOpen("c:\lawton\files\waveform\hendrix.jim"
                        NULL,
                        MMIO_READWRITE|
                        MMIO_DENYNONE)) == NULL)
   return(ERROR_FILE_NOT_FOUND);
/* Fill in acb with data object info */
acb.ulACBLen = sizeof(ACB_MMIO);
acb.ulObjType = ACBTYPE_MMIO;
acb.hmmio = hmmioIn;
/* Associate the waveform data object with the source handler */
if (rc = SpiAssociate(hstream,hidSource,&acb))
   return(rc);    /* error! */
/* Start the streaming */
if (rc = SpiStartStream(hstream, SPI_START_STREAM))
   return(rc);    /* error! */
/* Do other processing */

/* Create a semaphore and block this thread on the semaphore */
/* until the streaming is complete                           */
                  .
                  .
                  .
/* Destroy the stream  */
if (rc = SpiDestroyStream(hstream))
   return(rc);    /* error! */

/* Perform any other resource cleanup */

/*************************************************************/
/*               E v e n t   P r o c e d u r e               */
/*************************************************************/
RC EventProc(EVCB *pEVCB)
{
/* Handle events from the Sync/Stream Manager */
switch (pEVCB->ulType) {

   case EVENT_IMPLICIT_TYPE:
      switch (pEVCB->ulSubType) {
         case EVENT_EOS:     /* End of Stream */
            /* Clear the semaphore so that the main */
            /* routine can continue execution.      */
            break;
         case EVENT_ERROR:   /* Error occurred */
            /* flag error condition in main line application */
            break;
         case EVENT_STREAM_STOPPED:  /* Discard/Flush Stop */
            /* Do processing while stream stopped */
            break;
         case EVENT_SYNC_PREROLLED:  /* Stream is prerolled */
            /* Now that the producers are prerolled, do a real
               start to the stream handlers */
            break;
         default:
            /* Unknown event, do something */
         }
      break;
   default:
      /* Unknown event, do something */
   }
}
```

Synchronized MIDI and Waveform Streams

This scenario describes how two streams (waveform audio and MIDI) are created and synchronized with each other. The audio stream is the synchronization master stream. The source handler for the waveform stream is the File System Stream Handler DLL. The source handler for the MIDI stream is the File System Stream Handler DLL.

Currently, the audio stream handler does not provide the ability to be a slave to a master stream. It does, however, have the ability to be the master stream handler.

```
/************************************************************/
/*                        M A I N                           */
/************************************************************/
main()
{
RC           ulRC;             /* Error return code */

SPCBKEY      spcbkey;          /* Data type to stream */
IMPL_EVCB    evcb1, evcb2;     /* Event control block for implicit events */
ACB_MMIO     acb;              /* Associate control block used to assoc */
                               /* The file to stream (play) */
DCB_AUDIOSH  dcb;              /* Audio device driver information */
HAND         ahand[5];         /* Enumerate handler info */
ULONG        ulNumHand = 5;
HID          hidSource1, hidTarget1, hidunused;    /* Handler IDs */
HID          hidSource2, hidTarget2
HSTREAM      hstream1, hstream2; /* Stream handle */
HEVENT       hevent1, hevent2; /* Event handle for implicit events */
HMMIO        hmmio1, hmmio2;   /* Handle to mmio file open */
SLAVE        slave[1];         /* Sync slave structure */

/* Get list of all Stream handlers in system */
if (rc = SpiEnumerateHandlers(&ahand, &ulNumHand))
   return(rc);    /* error! */

/******************************************************/
/****** CREATE WAVEFORM AUDIO STREAM  (Stream # 1) ******/
/******************************************************/
/* Get the stream Handler ID for the file system stream handler */
if (rc = SpiGetHandler("FSSH",&hidSource1,&hidunused))
   return(rc);    /* error! */
/* Get the stream handler ID for the Audio Stream Handler */
if (rc = SpiGetHandler("AUDIOSH$",&hidunused,&hidTarget1))
   return(rc);    /* error! */
/* Create a data stream from the file system to */
/* the audio ACPA device connected to speakers.  */
spcbkey.ulDataType = DATATYPE_ADPCM_AVC;
spcbkey.ulDataSubType = ADPCM_AVC_HQ;
spcbkey.ulINtKey = 0;
strcpy(dcb.szDevName,"AUDIO1$");
dcb.ulDCBLen = sizeof(dcb);
dcb.hSysFileNum = (** hSysFileNum returned by audio dd on audio_init call)
if (rc = SpiCreateStream(hidSource1,
                         hidTarget1,
                         &spcbkey,
                         NULL,
                         (PDCB)&dcb,
                         (PEVCB)&evcb1,
                         EventProc,
                         0,
                         &hstream1,
                         &hevent1))
   return(rc);    /* error! */
/* USE MMIO to access a waveform audio object */
if ((hmmio1 = mmioOpen("E:\Mygreat\waveform\file\screams.hq"
                       NULL,
                       MMIO_READWRITE|
                       MMIO_DENYNONE)) == NULL)
   return(ERROR_FILE_NOT_FOUND);
/* Fill in acb with data object info */
acb.ulACBLen = sizeof(ACB_MMIO);
acb.ulObjType = ACBTYPE_MMIO;
```

```
acb.hmmio = hmmio1;
/* Associate the waveform data object with the source handler */
if (rc = SpiAssociate(hstream1,hidSource1,&acb))
   return(rc);    /* error! */

/*******************************************/
/****** CREATE MIDI STREAM  (Stream # 2) ******/
/*******************************************/
/* Get the stream handler ID for the File System Stream Handler */
if (rc = SpiGetHandler("FSSH",&hidSource2,&hidunused))
   return(rc);    /* error! */
/* Get the stream handler ID for the MIDI Audio Stream Handler */
if (rc = SpiGetHandler("AUDIOSH$",&hidunused,&hidTarget2))
   return(rc);    /* error! */

/* Create a data stream from the memory to an */
/* OEM audio device connected to speakers.     */
spcbkey.ulDataType = DATATYPE_MIDI;
spcbkey.ulDataSubType = SUBTYPE_NONE;
spcbkey.ulINtKey = 0;
strcpy(dcb.szDevName,"AUDIO99$");
dcb.ulDCBLen = sizeof(dcb);
dcb.hSysFileNum = (** hSysFileNum returned by audio dd on audio_init call)
if (rc = SpiCreateStream(hidSource2,
                         hidTarget2,
                         &spcbkey,
                         NULL,
                         (PDCB)&dcb,
                         (PEVCB)&evcb2,
                         EventProc,
                         0,
                         &hstream2,
                         &hevent2))
   return(rc);    /* error! */

/* USE MMIO to access a waveform audio object */
if ((hmmio2 = mmioOpen("E:Mygreat\midi\file\screams.mid"
                       NULL,
                       MMIO_READWRITE|
                       MMIO_DENYNONE)) == NULL)
   return(ERROR_FILE_NOT_FOUND);
/* Fill in acb with data object info */
acb.ulACBLen = sizeof(ACB_MMIO);
acb.ulObjType = ACBTYPE_MMIO;
acb.hmmio = hmmio2;
/* Associate the waveform data object with the source handler */
if (rc = SpiAssociate(hstream2,hidSource2,&acb))
   return(rc);    /* error! */


/*************************************************************/
/***** SETUP THE SYNCHRONIZATION BETWEEN THE TWO STREAMS ******/
/*************************************************************/
slave[0].hSlaveStream = hstream2;   /* MIDI stream is slave */
slave[0].mmtimeStart = 0;           /* Time to start slave */
if (rc = SpiEnableSync(hstream1,          /* Audio stream is master */
                       &slave,
                       1,                 /* Number of slaves */
                       0))                /* Use spcb sync granularity */
   return(rc);    /* error! */
/* Start streams by passing the handle to the stream sync master */
if (rc = SpiStartStream(hstream1, SPI_START_SLAVES))
   return(rc);    /* error! */

/* Do other processing */
                  .
                  .
                  .

/* Create a semaphore and block this thread on the semaphore */
/* until the streaming is complete                           */
                  .
                  .
                  .

/* Destroy both streams when completed */
if (rc = SpiDestroyStream(hstream1))
   return(rc);    /* error! */
if (rc = SpiDestroyStream(hstream2))
```

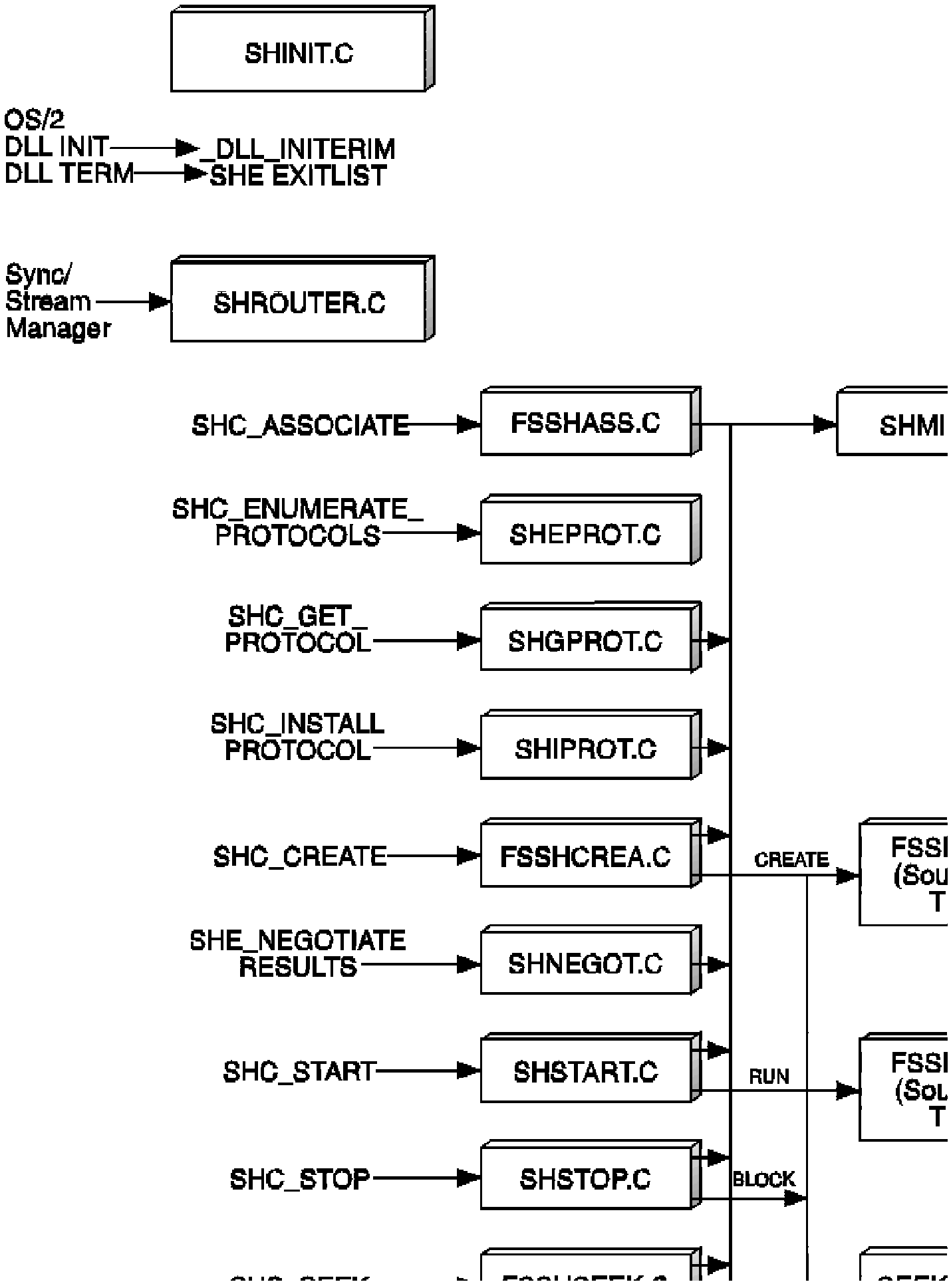
```
   return(rc);    /* error! */

/* Perform other resource cleanup */
}
/*********************************************************/
/*               E v e n t   P r o c e d u r e           */
/*********************************************************/
RC EventProc(EVCB *pEVCB)
{
if (pEVCB == &EVCB1)      /* Stream #1 or Stream # 2 event ? */
   /* Process Stream #1 events */
else
   /* Process Stream #2 events */
/* Handle events from the Sync/Stream Manager */
switch (pEVCB->ulType) {
   case EVENT_IMPLICIT_TYPE:
      switch (pEVCB->ulSubType) {
         case EVENT_EOS:     /* End of Stream */
            /* Clear the semaphore so that the main routine can */
            /* continue execution.                              */
            break;
         case EVENT_ERROR:   /* Error occurred */
            /* flag error condition in main line application */
            break;
         case EVENT_STREAM_STOPPED:  /* Discard/Flush Stop */
            /* Do processing while stream stopped */
            break;
         case EVENT_SYNC_PREROLLED:  /* Stream is prerolled */
            /* Now that the producers are prerolled, do a real
               start to the stream handlers */
            break;
         default:
            /* Unknown event, do something */
         }
      break;
   default:
      /* Unknown event, do something */
   }
}
```

DLL Model: File System Stream Handler

The multimedia system provides stream handlers at both system kernel level (Ring 0) as OS/2 PDDs, and at application level (Ring 3) as OS/2 DLLs. The reason for having both types of stream handlers is that while some streams are ideally controlled by a direct connection between the stream handler and a device's PDD, other streams are not associated with a data source or target, which maps physically to a specific device. For example, the File System Stream Handler is a DLL, because all file system I/O functions are available as Ring 3 OS/2 API, and service all file system devices. This eliminates the need to build a specific stream handler PDD for every device the file system can access.

File System Stream Handler Modules

The following figure illustrates the outline of the file system stream handler (FSSHT.DLL).


SHINIT.C
OS/2
DLL INIT
DLL TERM
_DLL_INITERIM
SHE EXITLIST
Sync/
Stream
Manager
SHROUTER.C
SHC_ASSOCIATE
FSSHASS.C
SHMI
SHC_ENUMERATE_
PROTOCOLS
SHEPROT.C
SHC_GET_
PROTOCOL
SHGPROT.C
SHC_INSTALL
PROTOCOL
SHIPROT.C
SHC_CREATE
FSSHCREA.C
CREATE
FSSI
(Sou
T
SHE_NEGOTIATE
RESULTS
SHNEGOT.C
SHC_START
SHSTART.C
RUN
FSSI
(Sou
T
SHC_STOP
SHSTOP.C
BLOCK

The descriptions for the sample file system stream handler modules shown in the example in section File System Stream Handler Modules are contained in the following table.

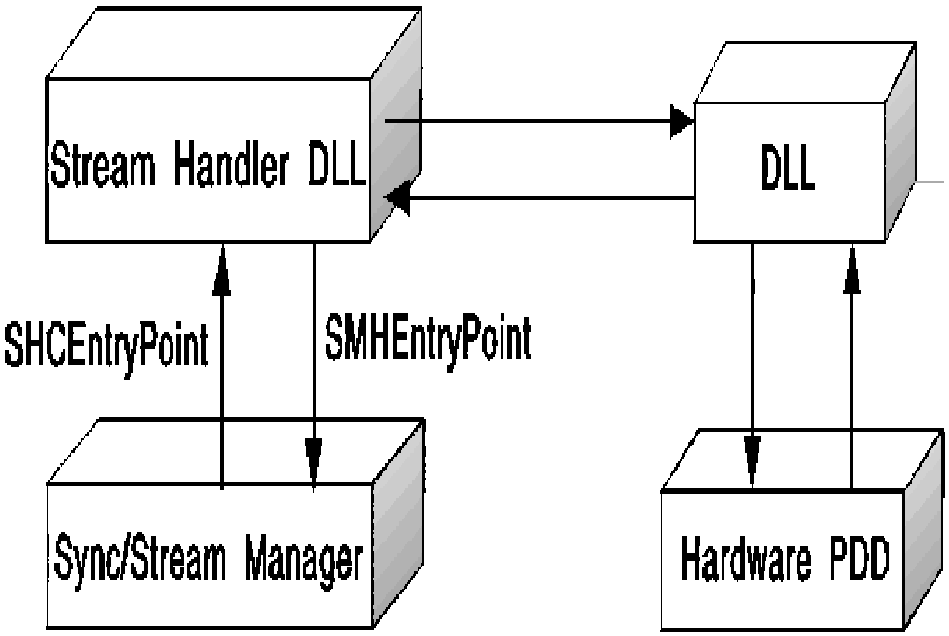
Procedure Name	Description
SHINIT.C	Receives control when FSSHT.DLL is loaded and performs the following functions: DLL_INITTERM Initialization routine to register the stream handler with the Sync/Stream Manager (SSM), allocate heap memory, and create mutex semaphores. SHEXITLIST Termination routine to deregister the stream handler, destroy any active streams, reload heap memory, and close mutex semaphores.
SHROUTER.C	Receives the stream handler command (SHC) messages from SSM and routes them to the appropriate low-level routines.
FSSHASS.C	Processes the SHC_ASSOCIATE stream handler command message to associate an object (file handler, CD ROM drive letter, and so on) with a data stream.
SHEPROT.C	Processes the SHC_ENUMERATE stream handler command message to return a list of stream protocols for a given stream.
SHGPROT.C	Processes the SHC_GET_PROTOCOL stream handler command message to return a stream protocol control block (SPCB) for a given data type.
SHIPROT.C	Processes the SHC_INSTALL_PROTOCOL stream handler command message to install or replace an SPCB for a given data type.
FSSHCREA.C	Processes the SHC_CREATE stream handler command message to create a stream instance for a given data type and subtype. If the create is for a source stream handler, a read thread in FSSHREAD.C is created. Otherwise a write thread is created in FSSHWRIT.C for a target stream handler.
SHNEGOT.C	Processes the SHC_NEGOTIATE_RESULTS stream handler command message to save the negotiated SPCB for a given stream instance.
SHSTART.C	Processes the SHC_START stream handler command message to start the data streaming for a given stream instance. For a source stream handler, the read thread in FSSHREAD.C is run. For a target stream handler, the write thread in FSSHWRIT.C is run.
SHSTOP.C	Processes the SHC_STOP stream handler command message to stop the data streaming for a given stream instance. For a source stream handler, the read thread in all the buffers is either discarded (STOP DISCARD) or returned to SSM (STOP FLUSH). For a target stream handler, the write thread in FSSHWRIT.C is blocked, with buffers either discarded or flushed.
FSSHSEEK.C	Processes the SHC_SEEK stream handler command message to seek to a specified point in the stream object. The seek can be from the beginning of the file, the current position, or the end o the file. The seek point can be specified in bytes offset or MMTIME units. This module interfaces with low-level routines in SEEKCALC.ASM to perform conversion from MMTIME values to bytes, and with mmioSeek to perform the actual seek in the object.
SEEKCALC.ASM	Performs the low-level conversions of MMTIME to bytes, and bytes to MMTIME, needed by FSSHSEEK.C.
SHDESTRY.C	Processes the SHC_DESTROY stream handler command

	message to remove a stream instance. Either the read thread in FSSHREAD.C or the write thread in FSSHWRIT.C is terminated when this message is received.
SHMISC.C	Supporting routines used by the other stream handler modules; for example, searching the stream instance chain to find a particular stream instance, finding an extended stream protocol control block (SPCB), and deallocating a stream instance.
FSSHREAD.C	Reads an object from the file system. A read thread is created when the SHC_CREATE message is received in FSSHCREA.C and destroyed when the SHC_DESTROY message is received in SHDESTRY.C. The read is accomplished by interfacing with a mmioRead routine and low-level routines in SHIOUTIL.C to check processing flags and report events to SSM.
SHIOUTIL.C	Low-level routines used by FSSHREAD.C and FSSHWRIT.C modules. This module performs checks of processing flags and also reports processing events to SSM.
FSSHDAT.C	Global data declaration for the stream handler
makefile	Makefile to build FSSHT.DLL sample file system stream handler

---

Entry Points Diagram

The Sync/Stream Manager DLL exports SPI services to higher-level OS/2 multimedia components (such as media drivers) and exports Stream Manager Helper (SMH) messages to support the stream handler DLLs. Additional SMH messages are exported by the SSM device driver to stream handler device drivers using standard OS/2 inter-device driver communication (IDC) interfaces (established using DevHelp_AttachDD).



SMHEntryPoint

The Stream Manager Helper (SMH) messages are exported for use by both device driver and DLL stream handlers through the SMHEntryPoint. The stream handlers use these helper routines to register with the Sync/Stream Manager, report events and synchronization cues, and request or return buffers. They are synchronous calls and are available to Ring 3 DLL stream handlers as a DLL call (Ring 0 stream handlers as an IDC call).

SHCEntryPoint

As illustrated in the following example, a DLL stream handler implements function through an additional entry point. DLL stream handlers must supply an interface to communicate with the Sync/Stream Manager. This interface, called the stream handler command (SHC) interface, exports stream handler commands (SHCs) for use by the Sync/Stream Manager to set up and control data streams. These SHCs are accessible through a single entry point, SHCEntryPoint.

```
RC ShcRouter(pshc)
PSHC_COMMON pshcNow;

{ /* Start of ShcRouter */

RC rc = NO_ERROR;                       /* local return code */
  switch (pshc->ulFunction)
  {
    case SHC_ASSOCIATE:
         {
           rc = ShcAssociate((PPARM_ASSOC)pshcNow);
           break;
         }
    case SHC_CREATE:
         {
           rc = ShcCreate((PPARM_CREATE)pshcNow);
           break;
         }
    case SHC_DESTROY:
         {
           rc = ShcDestroy((PPARM_DESTROY)pshcNow);
           break;
         }
    case SHC_SEEK:
         {
           rc = ShcSeek((PPARM_SEEK)pshcNow);
           break;
         }
    case SHC_START:
         {
           rc = ShcStart((PPARM_START)pshcNow);
           break;
         }
    case SHC_STOP:
         {
           rc = ShcStop((PPARM_STOP)pshcNow);
           break;
         }
    case SHC_GET_PROTOCOL:
         {
           rc = ShcGetProtocol((PPARM_GPROT)pshcNow);
           break;
         }
    case SHC_INSTALL_PROTOCOL:
         {
           rc = ShcInstallProtocol((PPARM_INSTPROT)pshcNow);
           break;
         }
    case SHC_ENUMERATE_PROTOCOLS:
         {
           rc = ShcEnumerateProtocols((PPARM_ENUMPROT)pshcNow);
           break;
         }
    case SHC_NEGOTIATE_RESULT:
         {
           rc = ShcNegotiateResult((PPARM_NEGOTIATE)pshcNow);
           break;
```

```
        }
    default:
        {
          rc = ERROR_INVALID_FUNCTION;
          break;
        }
  } /* endswitch */

  return(rc);

} /* End of ShcRouter */
```

DLL Initialization

The following example illustrates a DLL stream handler's initialization process. During initialization, the stream handler must perform any global DLL initialization the first time the DLL is loaded and perform any instance initialization for each instance of the stream handler DLL. Each instance of the DLL represents a process. The stream handler must register itself with the Sync/Stream Manager during global initialization. The stream handler should also register an exit-list routine if any is required for DLL termination.

```
ULONG _DLL_InitTerm( HMODULE hmod, ULONG fTerm)

{ /* Start of ShInit */

RC rc = NO_ERROR;                       /* local return code */
PARM_REG smhRegParm;                    /* Parameters for SMH_REGISTER */
int Registered = FALSE;
int HeapAllocated_Attached = FALSE;
int GlobDataMtxCreated = FALSE;
hmod;

  /*
   * Checking this parameter will insure that this routine will only
   * be run on an actual initialization.  Return success from the
   * termination.
   */

  if (fTerm)
     {
     return (1L);
     }

  if (_CRT_init())
     {
     return (0L);
     }

  /*
   * Get the semaphore controlling the process count and update the process
   * count if successful.  Then after we have the sem, if the count is 1,
   * we are guaranteed that we must do the global initializations.
   *
   * There is a problem.  To determine if we need to create or open the
   * semaphore, we need to check the ProcessCount to see if this is the
   * first process.  But since we don't have the semaphore, we don't have
   * a guarantee the count won't change.  If we get caught in this window
   * then we will fail to get the semaphore and the rc will indicate error.
  */

  if (ulProcessCount == 0)
    { /* First process */

      if (!(rc = DosCreateMutexSem(pszProcCntMutexName,
                                   &hmtxProcCnt,
                                   0L,
                                   TRUE)))
        {
          ulProcessCount++;
        }
    } /* First process */
```

```
else
  { /* not first process */
    /* if a process exits and decrements ProcessCount before we get  */
    /* the semaphore here, we will fail. */
    if (!(rc = DosOpenMutexSem(pszProcCntMutexName,
                               &hmtxProcCnt)))
      {
        if (!(rc = DosRequestMutexSem(hmtxProcCnt,
                                      SEM_INDEFINITE_WAIT)))
          {
            ulProcessCount++;
          }
        else
          {
            DosCloseMutexSem(hmtxProcCnt);
          }
      }
  } /* not first process */

if (!rc)
{ /* Semaphore ok, In critical section */
  /*
   * If this is the first time this init routine is called then we are
   * being brought in by the loader the first time, so we need to
   * register with the SSM.
   */
  if (ulProcessCount == 1)
    { /* First process */
      smhRegParm.ulFunction = SMH_REGISTER;
      smhRegParm.pszSHName = pszHandlerName;
      smhRegParm.phidSrc = &SrcHandlerID;
      smhRegParm.phidTgt = &TgtHandlerID;
      smhRegParm.pshcfnEntry = ( PSHCFN ) ShcRouter;

      smhRegParm.ulFlags = 0L;
      if (ulHandlerFlags & HANDLER_CAN_BE_SOURCE)
        {
          smhRegParm.ulFlags = REGISTER_SRC_HNDLR;
        }
      if (ulHandlerFlags & HANDLER_CAN_BE_TARGET)
        {
          smhRegParm.ulFlags |= REGISTER_TGT_HNDLR;
        }
      if (ulHandlerFlags & HANDLER_IS_NONSTREAMING)
        {
          smhRegParm.ulFlags |= REGISTER_NONSTREAMING;
        }
      rc = SMHEntryPoint((PVOID)&smhRegParm);
      /*
       * If ok then allocate our memory pool (heap), since it is under
       * semaphore control create and get the semaphore first.
       */
      if (!rc)
        { /* Register ok */
          Registered = TRUE;
          hHeap = HhpCreateHeap(HEAPSIZE,
                                HH_SHARED);
          if (hHeap)
            { /* Heap Allocated */
              HeapAllocated_Attached = TRUE;
              if (!(rc = DosCreateMutexSem(NULL,
                                           &hmtxGlobalData,
                                           DC_SEM_SHARED,
                                           FALSE)))
                {
                  GlobDataMtxCreated = TRUE;
                }
            } /* Heap Allocated */
          else
            { /* Heap not allocated */
              rc = ERROR_ALLOC_RESOURCES;
            } /* Heap not allocated */
        } /* Register ok */
    } /* First Process */
  else
    { /* Not first process */
      if (!(rc = DosOpenMutexSem(NULL,
                                 &hmtxGlobalData)))
        { /* Global data semaphore opened */
```

```
            GlobDataMtxCreated = TRUE;

            if (!ENTERCRIT(rc))
              { /* Global Data Sem obtained */

                if (HhpAccessHeap(hHeap, HhpGetPID()))
                  { /* Error accessing heap */
                    rc = ERROR_ALLOC_RESOURCES;
                  } /* Error accessing heap */
                else
                  { /* Heap attached */
                    HeapAllocated_Attached = TRUE;
                  } /* Heap attached */

                EXITCRIT;

              } /* Global Data Sem obtained */
          } /* Global data semaphore opened */
      } /* Not first process */

    /*
     * If things are ok, Register an exit list handler and if that works
     * increment the process count.
     */
    if (!rc)
      {
        rc = DosExitList(EXTLSTADD_ORDER,
                         (PFNEXITLIST)ShExitList);
      }

    if (rc)
      { /* Error occurred - Clean Up */
        if (HeapAllocated_Attached)
          {
            HhpReleaseHeap(hHeap,
                           HhpGetPID());
          }
        if (GlobDataMtxCreated)
          {
            DosCloseMutexSem(hmtxGlobalData);
          }
        if (Registered)
          { /* Deregister */
            PARM_DEREG smhDeregParm;

            smhDeregParm.ulFunction = SMH_DEREGISTER;
            smhDeregParm.pszSHName = pszHandlerName;
            SMHEntryPoint(&smhDeregParm);
          } /* Deregister */
        ulProcessCount--;
        DosReleaseMutexSem(hmtxProcCnt);
        DosCloseMutexSem(hmtxProcCnt);
      } /* Error occurred - Clean Up */
    else
      {
#ifdef MMRAS_PTRACE
        InitPTrace()
#endif
        DosReleaseMutexSem(hmtxProcCnt);
      }

  } /* Semaphore ok, In critical section */
  /*
   * The return codes expected are:
   *  TRUE (any non-zero) - init routine worked
   *  FALSE (zero)        - init routine failed
   *
   * So we have to reverse the local rc before we pass it back.
   */
  return(!rc);

} /* End of SHInit */
```

The stream event handling logic in the DLL stream handlers is essentially the same as in device driver stream handlers. The only significant difference is that stream events are detected and reported at task time in a DLL, rather than at interrupt time as in the case with device drivers

```
#include        "os2.h"
#include        "os2me.h"

  RC            rc;                     /* Error return code            */
  HID           hidSource;              /* Source handler ID            */
  TIME_EVCB     timeevcb;               /* Cuepoint event control block */
  HEVENT        hevent;                 /* time event handle            */
  HSTREAM       hstream;                /* Stream handle                */
  HID           hid;                    /* Handler ID                   */
  PARM_EVENT    parm_event;             /* Report event parameter block */
  MMTIME        mmtimeCurrent;          /* Current stream time          */
  PSMHFN        SMHEntryPoint;          /* Pointer to SMH entry point   */

/*--------------------------------------------------------------------*/
/*  Report an event.                                                  */
/*--------------------------------------------------------------------*/
  parm_event.ulFunction = SMH_REPORTEVENT;     /* Set function         */
  parm_event.hid = hidSource;                  /* Source handler ID    */
  parm_event.hevent = hevent;                  /* Event handle         */
  parm_event.pevcbEvent = (PEVCB) &timeevcb;   /* Pointer to Time EVCB */

  timeevcb.ulType = EVENT_CUE_TIME;            /* Set event type       */
  timeevcb.hstream = hstream;                  /* Set stream handle    */
  timeevcb.hid = hid;                          /* Set handler ID       */
  timeevcb.ulStatus = 0;                       /* No status            */
  timeevcb.mmtimeStream = mmtimeCurrent;       /* Set current time     */

  if (rc = SMHEntryPoint (&parm_event))
     return(rc);                               /* Error! */
```

Synchronization

DLL stream handlers can process sync pulses as either a master or slave stream. A master of a sync group must have the SPCBHAND_GENSYNC or the SPCBHAND_NOSYNC bits set in the SPCB that is returned to the SSM on the SHC_CREATE call. Otherwise, the stream handler cannot be a master.

If you would like to create a stream handler that can generate sync pulses, you must either use a hardware device (such as an audio card) to generate timing interrupts at a fixed rate, or use the OS/2 timer services such as the OS/2 device driver SetTimer DevHelp or the OS/2 DosAsyncTimer functions. You can then call GetDOSVar to query the address of the global information segment structure (GlobalInfoSeg). This allows you to get the real time so that you can detect missed timing messages. For further information, refer to the *Physical Device Driver Reference for OS/2* .

Note: We provide the architecture for synchronization. The audio stream handler can be a master (generate sync pulses). The audio stream handler can also be a slave, but it does not support the ability to re-synchronize to a master stream.

```
/*********************************************************************/
/* Master                                                            */
/* Synchronization - If we are the master stream, then report the    */
/*                   current time to the Sync/Stream Manager so that */
/*                   it can be distributed to any slave streams.     */
/*                                                                   */
/************************ END OF SPECIFICATIONS ************************/
PARM_EVENT  parmevent;              /* parms for SMH_NOTIFY calls    */
SYNC_EVCB   syncevcb;               /* Cuepoint event control block  */

if (psib->ulStatus & STREAM_MASTER_SYNC) {
   parm_event.ulFunction = SMH_REPORTEVENT;     /* Set function         */
   parm_event.hid = psib->HandlerID;            /* Source handler ID    */
   parm_event.hevent = hevent;                  /* Event handle         */
   parm_event.pevcbEvent = (PEVCB) &syncevcb;   /* Pointer to Sync EVCB */
   syncevcb.ulType = EVENT_CUE_TIME;            /* Set event type       */
   syncevcb.hstream = psib->hStream;            /* Set stream handle    */
   syncevcb.hid = hid;                          /* Set handler ID       */
   syncevcb.ulStatus = 0;                       /* No status            */
```

```
   syncevcb.mmtimeStream = mmtimeCurrent;       /* Set current time     */

   if (rc = SMHEntryPoint (&parm_event))
      return(rc);                               /* Error! */

   }

/*********************************************************************/
/* Slave                                                             */
/* Synchronization - If we are the slave stream, then update the slave */
/*                   time and determine if this slave stream is      */
/*                   too slow or too fast with respect to the master   */
/*                   stream.                                         */
/*                                                                   */
/*********************** END OF SPECIFICATIONS ***********************/

if ((pSTREAM)->ulStateFlg & STREAM_SLAVE_SYNC) {
        (pSTREAM)->SyncEvcb.mmtimeSlave = mmtimeCurrStreamTime;
        if ((pSTREAM)->SyncEvcb.ulStatus & SYNCPOLLING) {
             if (mmtimeCurrStreamTime > (pSTREAM)->SyncEvcb.mmtimeMaster)
                     /* I need to slow my stream */
                     ;
             if (mmtimeCurrStreamTime < (pSTREAM)->SyncEvcb.mmtimeMaster)
                     /* I need to speed up my stream */
                     ;
        }
```

Worker Thread Creation

Stream handler DLLs create worker threads for each stream created. A worker thread, for example, would consist of code to request a buffer from the Sync/Stream Manager, fill it with data from a device and return it to the Sync/Stream Manager. This would continue until the end of stream or until some other kind of stream stop.

A worker thread does all the work of the stream handler itself. It is a good idea to create a worker thread for each stream instance. Basically, a worker thread either loops in the read routine or loops in the write routine. This depends on whether it is a source or target. If it is a source, the thread loops in the read routine and reads from its device using whatever commands it uses to interface with its device. It also interfaces with the Sync/Stream Manager requesting empty buffers and returning full buffers. On the other hand, when the DLL stream handler is the target, the thread will also be in a big loop, but it is performing different operations. It will be requesting full buffers from the Sync/Stream Manager and then consuming those buffers by passing them off to the device (in whatever way it communicates with its device).

```
#include <os2.h>
#include <os2me.h>
#include <hhpheap.h>
#include <shi.h>

PSIB psib;                              /* Stream Instance Block */

{ /* Start of FsshRead */

RC          rc = NO_ERROR;              /* local return code */
char        NeedBuf;                    /* local loop Boolean */
LONG        NumBytesIO;                 /* Number of bytes from mmio */
PARM_NOTIFY npget, npret;               /* parms for SMH_NOTIFY calls */
SRCBUFTAB   SrcBufTab = {0};            /* Source buffer table */
ULONG       ulStopped = DO_SLEEP;       /* did we get stop disc or flush */
BOOL        bAtEOS = FALSE;             /* End of Stream indicator */
ULONG       ulPostCount;                /* Temp to hold count */

  /* Before we start lets do some init stuff: */

  npget.ulFunction = SMH_NOTIFY;
  npget.hid = psib->HandlerID;
  npget.hstream = psib->hStream;
  npget.ulGetNumEntries = 1L;
  npget.ulRetNumEntries = 0L;
  npget.pGetBufTab = &SrcBufTab;
  npget.pRetBufTab = NULL;

  npret.ulFunction = SMH_NOTIFY;
```

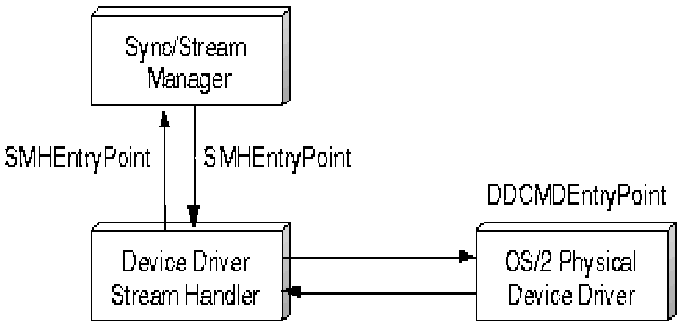
```
npret.hid = psib->HandlerID;
npret.hstream = psib->hStream;
npret.ulFlags = BUF_RETURNFULL;
npret.ulGetNumEntries = 0L;
npret.ulRetNumEntries = 1L;
npret.pGetBufTab = NULL;
npret.pRetBufTab = &SrcBufTab;

/* Wait until we get the ShcStart */

DosWaitEventSem(psib->hevStop, SEM_INDEFINITE_WAIT);

/* We will loop forever getting an empty buffer, calling the device to */
/* fill up the buffer, sending it to the consumer.  During each        */
/* iteration of the loop we will check the action flags for            */
/* asynchronous requests to do things.                                 */

if (psib->ulActionFlags & SIBACTFLG_KILL)
  { /* Must have been a create error */
    rc = 1L;
  } /* Must have been a create error */

/*  Start the main loop */

while (!rc)
  { /* while no severe error */

  if (psib->ulActionFlags)
    rc = CheckNSleep(psib);

  /*
   * Get a buffer
   */
  NeedBuf = TRUE;
  while ((NeedBuf) && (!rc))
    { /* while we don't have a buffer */

    /* Clear the stop sem, so if after we call ssm to get a buffer if */
    /* it returns none avail then we won't miss a SSMBuffer Start     */
    /* before we go to sleep.                                         */
      DosResetEventSem(psib->hevStop, &ulPostCount);

      npget.ulFlags = BUF_GETEMPTY;
      rc = SMHEntryPoint(&npget); /* get a buffer */
      if (!rc)
        {
          NeedBuf = FALSE;
       /* make sure attribute is 0 so we don't pass around a bad value */
          SrcBufTab.ulMessageParm = 0L;
        }
      else
        { /* return code from smhnotify */
          if (rc == ERROR_BUFFER_NOT_AVAILABLE)
            { /* buffer not available */

          /* the smhnotify resets the num entries to 0 when none avail */
              npget.ulGetNumEntries = 1L;

              ulStopped = DO_SLEEP;
              rc = SleepNCheck(psib, &ulStopped);

            } /* buffer not available */
        } /* return code from smhnotify */
    } /* while we don't have a buffer */

  /* We have a buffer or an error */

  if (!rc)
    { /* have a buffer - do the read */

      NumBytesIO = mmioRead((HMMIO)psib->ulAssocP1,
                            (PCHAR)SrcBufTab.pBuffer,
                            (LONG)SrcBufTab.ulLength);

      if (NumBytesIO == -1L)
        { /* an error */

          SrcBufTab.ulLength = 0L;
          /* get the real error code */
```

```
            rc = mmioGetLastError((HMMIO)psib->ulAssocP1);

            rc = ShIOError(psib, npret, rc);

          } /* an error */
        else
          { /* We have some data */

            if (NumBytesIO != (LONG)SrcBufTab.ulLength)
              { /* End of stream */
                npret.ulFlags |= BUF_EOS;
                bAtEOS = TRUE;
                DosResetEventSem(psib->hevStop, &ulPostCount);
                SrcBufTab.ulLength = NumBytesIO;
              } /* End of stream */

            /* Send the data to the stream manager */

            rc = SMHEntryPoint(&npret);
            if (!rc)
              { /* data sent ok */
                if (bAtEOS)
                  {
                    bAtEOS = FALSE;
                    ulStopped = DO_SLEEP;
                    rc = SleepNCheck(psib, &ulStopped);
                  }
              } /* data sent ok */
          } /* We have some data */

        /* Clear the EOS if it was set. And attribute */
        npret.ulFlags = BUF_RETURNFULL;
        SrcBufTab.ulMessageParm = 0L;

      } /* have a buffer - do the read */

    } /* while no severe error */

  /* We get here if an error has occurred or a kill has */
  /*  been sent.  In the case of the kill, reset the */
  /*  return code to 0 (no error) and exit the thread. */
  /*  Otherwise, report the error event and exit the */
  /*  thread. */

  if (psib->ulActionFlags & SIBACTFLG_KILL)
    {
      rc = 0L;
    }
  else
    {
      ReportEvent(psib,
                  rc,                  /* Return code */
                  EVENT_ERROR,         /* event type */
                  0L,                  /* user info */
                  NONRECOVERABLE_ERROR);  /* Severe Error */
    }

  /* Only set this flag when we no longer need access to the sib since */
  /* Destroy may get control and Free the sib. */

  psib->ulActionFlags |= SIBACTFLG_THREAD_DEAD;
  return;

} /* End of FsshRead */
```

Device Driver Model: Video PDD

A device driver stream handler is used as a source handler at the Ring 0 level. It interfaces with the Sync/Stream Manager to get or return stream data buffers. The stream handler also interfaces with the hardware's physical device driver (PDD) to give or receive data buffer pointers. The purpose of the stream handler is to alleviate the media driver from the task of sending a data stream to the PDD. Instead, the

media driver issues function calls to the SSM to initiate a stream. In turn, the SSM requests the stream handlers to regulate the proper stream flow without intervention from the application.



Device driver stream handlers include two main entry points - SMHEntryPoint and DDCMDEntryPoint.

SMHEntryPoint

Device driver stream handlers send Stream Manager Helper (SMH) routines to the SSM through the SMHEntryPoint. The stream handlers use these helper routines to register with the Sync/Stream Manager, report events and synchronization cues, and to request or return buffers. This interface is created using the standard inter-device driver communication (IDC) approach, established by the AttachDD DevHelp function during initialization processing.

DDCMDEntryPoint

Device driver stream handlers communicate with the hardware PDD through the DDCMDEntryPoint. This is accomplished through the use of device driver commands (DDCMDs), which allow a stream handler to request a PDD to perform functions such as starting, stopping, or resuming a device. This interface uses the IDC mechanism provided by the ATTACHDD DevHelp function. The stream handler must attach to the device driver to receive the DDCMD entry point address of the device driver. This function is performed before issuing device driver commands.

SHCEntryPoint

As illustrated in the following example, a device driver stream handler implements function through two additional entry points: SHCEntryPoint and SHDEntryPoint.

Device driver stream handlers receive commands from the Sync/Stream Manager to initialize and perform streaming functions. These stream handler commands (SHCs) are accessible through a single entry point, SHCEntryPoint. The main routine is an IDC interface with the Sync/Stream Manager Device Driver. The SSM calls the device driver stream handler by issuing stream programming interface (SPI) functions such as SpiCreateStream, SpiStartStream, and SpiStopStream.

The following example illustrates the code implementation of the SHCEntryPoint.

```
RC  DDCMDEntryPoint(PDDCMDCOMMON pCommon); /* PDD entry point from SH */
RC  SHDEntryPoint(PSHD_COMMON pCommon);    /* SH entry point from PDD */
RC  SHCEntryPoint(PSHC_COMMON pCommon);    /* SH entry point from SSM */
RC  SMHEntryPoint(PSHC_COMMON pCommon);    /* SSM entry point from SH */

ULONG   (*ShcFuncs[])(PVOID pCommon) = {   /* SHC function jump table */
                SHCAssociate,                   /* 0 */
                SHCClose,                       /* 1 */
                SHCCreate,                      /* 2 */
                SHCDestroy,                     /* 3 */
                SHCStart,                       /* 4 */
```

```
                SHCStop,                        /* 5 */
                SHCSeek,                        /* 6 */
                SHCEnableEvent,                 /* 7 */
                SHCDisableEvent,                /* 8 */
                SHCEnableSync,                  /* 9 */
                SHCDisableSync,                 /* 10 */
                SHCGetTime,                     /* 11 */
                SHCGetProtocol,                 /* 12 */
                SHCInstallProtocol,             /* 13 */
                SHCEnumerateProtocols,          /* 14 */
                SHCNegotiateResult              /* 15 */
                };
USHORT  MaxShcFuncs = sizeof(ShcFuncs)/sizeof(USHORT);

/*********************************************************************/

RC      SHCEntryPoint(pCommon)
PSHC_COMMON     pCommon;
{
      if (pCommon->ulFunction > (ULONG)MaxShcFuncs)
            return(ERROR_INVALID_FUNCTION);
                                        /* Check for valid function */

return(ShcFuncs[pCommon->ulFunction](pCommon));
}                                       /* Call SHC message         */

/*********************************************************************/

RC SHCStart(PPARM_START pStart)
{
   PSTREAM         pSTREAM;             /* Ptr to current stream */
   DDCMDREADWRITE  DDCMDReadWrite;
   DDCMDCONTROL    DDCMDControl;
   ulRC            rc;                  /* Return code           */


   if (rc = GetStreamEntry(&pSTREAM, pStart->hstream))
               return(rc);

   EnterCritSec;                         /* Disable interrupts */


   switch (pSTREAM->ulStateFlg) {
     case STREAM_RUNNING:
         /*********************************************************/
         /* Get a full buffer for playback                        */
         /*********************************************************/
         ParmNotify.ulFunction = SMH_NOTIFY;
         ParmNotify.hid = pSTREAM->hid;
         ParmNotify.hstream = pSTREAM->hStream;
         ParmNotify.ulGetNumEntries = 1;
         ParmNotify.ulRetNumEntries = 0;
         ParmNotify.pGetBufTab = &(pSTREAM->BufTab.aBufTab[usBufTabIndex]);
         ParmNotify.pRetBufTab = NULL;
         ParmNotify.ulFlags = BUF_GETFULL;
         SMHEntryPoint(&ParmNotify);

         /*********************************************************/
         /* Send full buffer to device (PDD)                      */
         /*********************************************************/

         DDCMDReadWrite.hStream = (*ppSTREAM)->hStream;
         DDCMDReadWrite.ulFunction = DDCMD_WRITE;
         DDCMDReadWrite.pBuffer = pCurrentBufTab->pBuffer;
         DDCMDReadWrite.ulBufferSize = pCurrentBufTab->ulLength;
         rc = (*pSTREAM->pDDCMDEntryPoint)(&DDCMDReadWrite);

         /*********************************************************/
         /* Start the device                                      */
         /*********************************************************/

         DDCMDControl.ulFunction = DDCMD_CONTROL;
         DDCMDControl.hStream = pSTREAM->hStream;
         DDCMDControl.pParm = NULL;
         DDCMDControl.ulParmSize = NULL;
         DDCMDControl.ulCmd = DDCMD_START;
         rc = (*pSTREAM->pDDCMDEntryPoint)(&DDCMDControl);
         break;
```

```
    }
    ExitCritSec;
        return(rc);
}
```

SHDEntryPoint

Device driver stream handlers receive commands from PDDs to report events and interrupts. These stream handler device (SHD) helper commands are provided through the SHDEntryPoint. This entry point is specifically used for the PDD to call back to the stream handler. For example, the PDD can send an SHD_REPORT_INT command to the stream handler to report status, indicate that a buffer is full, or specify that an additional buffer is required.

The following example illustrates the code implementation of the SHDEntryPoint.

```
ULONG   (*ShdFuncs[])(PVOID pCommon) = {  /* SHD message jump table */
                        SHDReportInt,                   /* 0 */
                        SHDReportEvent                  /* 1 */
                        };

USHORT  MaxShdFuncs = sizeof(ShdFuncs)/sizeof(USHORT);
/*****************************************************************************/

RC SHDEntryPoint(pCommon)
PSHD_COMMON     pCommon;
{
      if (pCommon->ulFunction > (ULONG)MaxShdFuncs)

             return(ERROR_INVALID_FUNCTION); /* Check for valid function */

      return(ShdFuncs[pCommon->ulFunction](pCommon));
}                                       /* Call SHC message         */

/*****************************************************************************/
```

Event Detection

A device driver stream handler must be able to keep track of time to detect events. Timing is important to know when the stream starts and stops, and when to report certain cue time events. There are two methods you can implement to detect events. They are:

- Request the PDD to monitor the time for use.
- Implement an algorithm that constantly monitors real-time.

PDD Monitors the Time

The first and most accurate method of event detection is to retrieve the time from the actual physical device. Interfaces are provided through the use of DDCMD_STATUS, which requests the current stream time from the physical device driver. When an application requests an event, the stream handler issues a DDCMD_STATUS command to the PDD to detect the event time. (There are also additional DDCMD messages that allow the PDD to inform the stream handler of the time. Refer to the *OS/2 Multimedia Programming Reference* for details.) When the time arrives, the PDD calls back to the stream handler through the SHDEntryPoint using the SHD_REPORT_EVENT message. At this time, the stream handler looks through its table, and identifies which event has now come due. In turn, the stream handler reports the event through the SMHEntryPoint on the SMH_REPORTEVENT call to the Sync/Stream Manager. Once received, the SSM reports the event back to the application as shown in the following example.

```
RC  SHDReportEvent(PSHD_REPORTEVENT pRptEvent)
{
   PSTREAM         pSTREAM;
   ulRC            rc;

   if (rc = GetStreamEntry(&pSTREAM, pRptEvent->hStream))
           return(rc);

   pSTREAM->ulStreamTime = pRptEvent->ulStreamTime;
   /* Update stream time */

   /*********************************************************/
   /* PDD detected an event and notified the stream handler */
   /*********************************************************/

   while (pEVENT != NULL) {
      if (pEVENT->hEvent == pRptEvent->hEvent) {
                  RptEvent.ulFunction = SMH_REPORTEVENT;
                  RptEvent.hid = pSTREAM->hid;
                  RptEvent.hevent = pEVENT->hEvent;
                  RptEvent.pevcbEvent = &(pEVENT->evcb);
                  /************************************/
                  /* call SSM to report event arrival */
                  /************************************/
                  VideoSH.pSMHEntryPoint(&RptEvent);     /* report it */
                  break;   /* process only one event and break out */
     }
         pEVENT = pEVENT->pNext;

   } /* end while */

   return(NO_ERROR);
}
```

Stream Handler Monitors the Time

The second method of event detection is for the stream handler to monitor the time, independent of the PDD. This method is less accurate and not as efficient as requesting the PDD to monitor the time. It involves implementing an algorithm that constantly monitors real-time. When the stream handler detects the appropriate time, it reports the event to the Sync/Stream Manager as shown in the following example.

```
RC  SHDReportEvent(PSHD_REPORTEVENT pRptEvent)
{
  PSTREAM         pSTREAM;
  ulRC            rc;

  if (rc = GetStreamEntry(&pSTREAM, pRptEvent->hStream))
          return(rc);

   pSTREAM->ulStreamTime = pRptEvent->ulStreamTime;
   /* Update stream time */

  /*************************************************************/
  /* Stream handler detected event and notified the SSM.       */
  /*************************************************************/

  while (pEVENT != NULL) {            /* process all possible events */
     if (pEVENT->hEvent == pRptEvent->hEvent) {
                  /*****************************************/
                  /* Poll the current time to determine if the */
                  /* event time is now.                    */
                  /*****************************************/
                  if (mmtimeCurrTime >= pTimeEvcb->mmtimeStream) {
                      pTimeEvcb->mmtimeStream = mmtimeCurrTime;
                      RptEvent.ulFunction = SMH_REPORTEVENT;
                      RptEvent.hid = pSTREAM->hid;
                      RptEvent.hevent = pEVENT->hEvent;
                      RptEvent.pevcbEvent = &(pEVENT->evcb);
                      /*************************************/
```

```
                    /* Call the Sync/Stream Manager to report */
                    /*  event arrival.                        */
                    /****************************************/
                    VideoSH.pSMHEntryPoint(&RptEvent); /* report it */
   }
     pEVENT = pEVENT->pNext;

  } /* end while */

  return(NO_ERROR);
}
```

Note that when you develop a stream handler, the stream handler should be able to support all time formats.

Note: Most physical device drivers use milliseconds as their reference; however, cuepoints initiated by the application come in as a request to the stream handler in the time format, MMTIME. Therefore, the stream handler must be able to convert from MMTIME to milliseconds and vice versa.

Cuepoints

Cuepoints remain active for the life of the stream-meaning that once enabled, they remain for the duration of the stream, whether the user decides to seek forwards or backwards. If the user seeks over an event, the stream handler should not detect that event. If the user seeks backwards, the stream handler should re-enable that event.

Error Detection

Stream handlers can detect errors and report them back to the Sync/Stream Manager to be passed back to the media driver. Once a physical device driver detects an error, the PDD must report the error to the stream handler through the SHDEntryPoint. In turn, the stream handler takes the appropriate action. If the error is a hard error, the stream handler must stop the device and report an event to the Sync/Stream Manager. The stream handler indicates that a hard error took place by referencing the error code number. For normal events, such as underruns and overruns, the target stream handler will attempt to get another buffer and then pause its device. The target stream handler will be restarted when more data is available to be output. This condition results in a break in the output data stream. Interleaved data format can cause underruns to occur when the end of the data is reached, but end of file has not been reached.

The code sample in the following example illustrates the code implementation of a stream handler detecting an error.

```
RptEvent.ulFunction = SMH_REPORTEVENT;
RptEvent.hid = pSTREAM->hid;
RptEvent.hevent = 0;                          /* must be 0 */
RptEvent.pevcbEvent = (PEVCB)&evcb;

evcb.ulType = EVENT_IMPLICIT_TYPE;              /* SPI event */
evcb.ulSubType = EVENT_ERROR;                   /* event type */
evcb.ulFlags = 0;
evcb.ulStatus = EVENT_ERROR;                    /* error code */
evcb.hstream = pSTREAM->hStream;
/*********************************/
/* call SSM to report event arrival*/
/*********************************/
VideoSH.pSMHEntryPoint(&Ramp.ptEvent);              /* report it! */
```

Synchronization

Each stream handler may, or may not, be able to generate or receive sync pulses. This capability for each stream handler is defined in the

SPCB for that stream handler. Synchronization pulses are passed as an event from the master stream handler.

Synchronization pulses are distributed by the Sync/Stream Manager based on the synchronization relationship of the programmed stream. This distribution is effective for both DLL and device driver stream handlers. Device driver stream handlers receive sync pulses through their sync pulse SYNC_EVCB. Each slave stream handler must regularly update the sync pulse SYNC_EVCB with its calculated stream time. The Sync/Stream Manager checks this slave-handler stream time against the master stream time and decides whether to send a sync pulse to this handler. The device driver stream handler checks for sync pulses from the Sync/Stream Manager by polling a flag in the sync pulse SYNC_EVCB. The Sync/Stream Manager sets the flag to indicate a sync pulse and updates the current master stream time. Typically, the PDD slave handler polls the flag once during interrupt processing and adjusts the stream usage accordingly. Refer to the following example for an example of how device driver stream handlers support synchronization.

```
/*************************************************************/
/* If we are the master, then report the time so that the slaves */
/* can adjust their time and be in sync with the master.         */
/*************************************************************/

if ((pSTREAM)->ulStateFlg & STREAM_MASTER_SYNC) {
        (pSTREAM)->SyncEvcb.mmtimeMaster = mmtimeCurrStreamTime;
        RptEvent.ulFunction = SMH_REPORTEVENT;
        RptEvent.hid = (pSTREAM)->hid;
        RptEvent.hevent = 0;          /* must be 0 for implicit events */
        RptEvent.pevcbEvent = (PEVCB)&((pSTREAM)->SyncEvcb);
        VideoSH.pSMHEntryPoint(&RptEvent); /* report master time */
}
/*************************************************************/
/* If we are the slave, then update slave time               */
/* and determine if this slave stream is too slow or too fast */
/* with respect to the master.                               */
/*************************************************************/

if ((pSTREAM)->ulStateFlg & STREAM_SLAVE_SYNC) {
        (pSTREAM)->SyncEvcb.mmtimeSlave = mmtimeCurrStreamTime;
        if ((pSTREAM)->SyncEvcb.ulStatus & SYNCPOLLING) {
             if (mmtimeCurrStreamTime > (pSTREAM)->SyncEvcb.mmtimeMaster)
                     /* I need to slow my stream */
                     ;
             if (mmtimeCurrStreamTime < (pSTREAM)->SyncEvcb.mmtimeMaster)
                     /* I need to speed up my stream */
                     ;
        }
```

Note: See Synchronization Features for more information on sync pulse generation and processing.

Inter-device Driver Communications (IDC)

The OS/2 multimedia Audio Device Driver is a standard OS/2 PDD that makes use of two OS/2 interfaces to define a standard for IBM audio device drivers: the audio IOCtl interface and the inter-device driver communication (IDC) interface. These interfaces provide the foundation for sharing of the audio device by OS/2 multimedia applications and media drivers.

The audio IOCtl interface is provided as subfunctions of the OS/2 DosDevIOCtl function. Media drivers running at Ring 3 privilege level use the IOCtl interface to set up the audio device for access, to change volume controls, and so on. The IOCtl interface is not meant to be used to send audio data to the device driver.

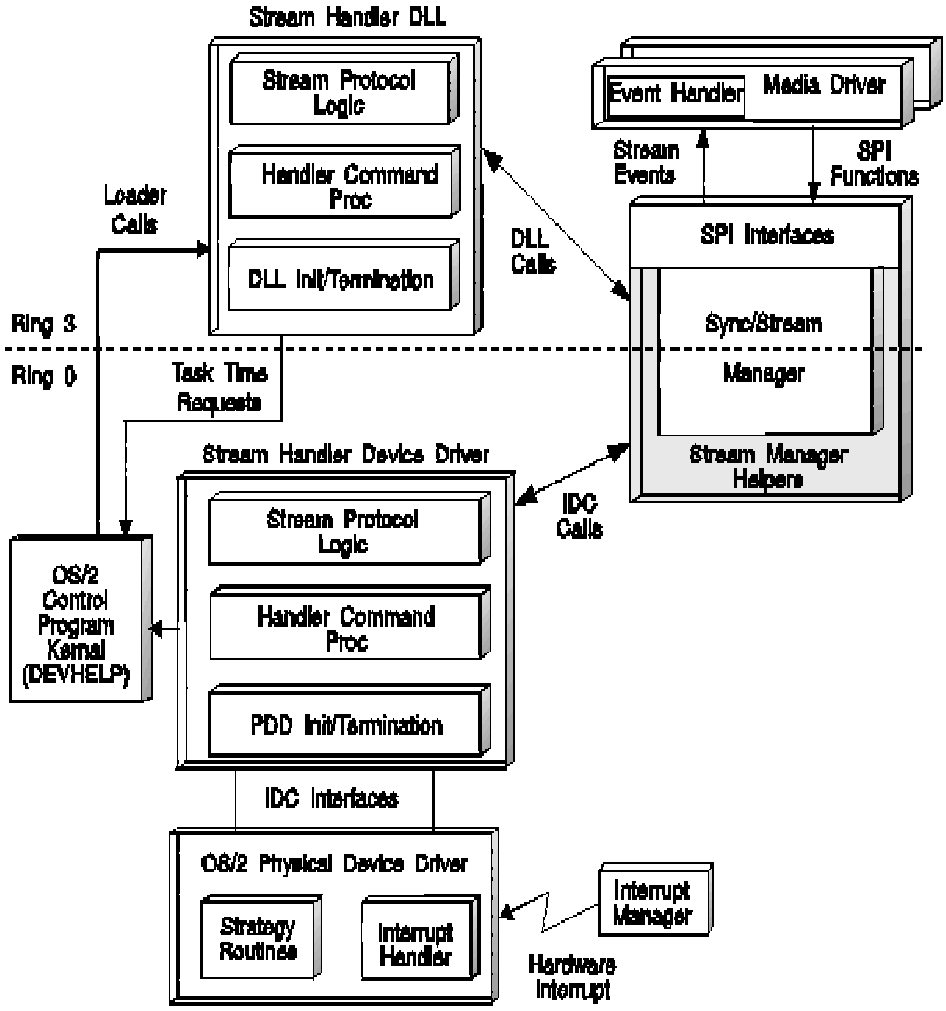
The IDC interface is provided by the OS/2 ATTACHDD DevHelp function. Stream handler device drivers and audio physical device drivers use the IDC interface to communicate with one another to coordinate their efforts to stream data to the audio device.

IDC Interface

Two entry points are provided for the two device drivers participating in the IDC interface: DDCMDEntryPoint and SHDEntryPoint. DDCMDEntryPoint allows the stream handler device driver to communicate to the PDD, and SHDEntryPoint allows the audio PDD to

communicate to the stream handler.

The audio PDD modules are shown in the following figure. Note that streaming data buffer pointers are passed by SSM to the Audio Stream Handler by means of SMH calls. Then the Audio Stream Handler passes pointers to the PDD using DDCMD messages.



Communication to the PDD

Seven messages are defined from the DDCMD entry point. Each message represents a specific request by the stream handler for the audio PDD to do perform an action.

Read and write messages allow the stream handler to get and receive data directly from the PDD without any intervention required by the application. The application neither has to send data through an IOCtl nor allocate and lock memory to perform data transfers.

Note: Refer to the Device Driver Command (DDCMD) messages in the *OS/2 Multimedia Programming Reference* for further information.

Communication to the Stream Handler

The SHDEntryPoint contains the following two messages. These messages are located in the SHDD.H file of the \TOOLKIT\H subdirectory.

SHDD.H contains the data structures, their type definitions, and #define statements for certain values. Note that the messages pass pointers to packets of data, to allow maximum flexibility for the future.

SHD_REPORT_INT
: The PDD uses this message when it needs things at interrupt time. For example, it uses this message to tell the stream handler it has used up all the data and needs some more.

  When the stream handler gets the call, it knows the PDD is passing back a buffer that it might already have consumed. So the stream handler returns on that call, giving the PDD a fresh buffer to consume.

SHD_REPORT_EVENT
: The stream handler uses this message to keep in sync with what the PDD is doing. For example, the stream handler can request the PDD to report back every tenth of a second that data is played. And the stream handler has all the logic to handle these events. The PDD examines the request, and during its checks when it realizes a tenth of a second has been played in data, the PDD calls SHD_REPORT_EVENT. The stream handler can do what it wants at this point, and the PDD returns.

  The PDD is the only one that really knows what is going on. In other words, only the PDD knows how much data, to the millisecond, has been played out to the device. The stream handler can approximate the data played, using calculations based on how much data has gone by. But the stream handler cannot calculate the data played to the millisecond, or even to the fraction of a millisecond, the way the PDD can.

Stream Handler Values

There are certain values that the stream handler looks for. For example, when the stream handler requests a stop or a pause on a DDCMD_CONTROL message, the pointer that comes back to the stream handler is a pointer to the cumulative time that the PDD has recorded in real time. So whenever the stream handler requests the device to stop, the PDD honors that request and tells the stream handler the real time the PDD stopped within the stream.

Another value the stream handler looks for is returned on DDCMD_STATUS. This is also a pointer to the cumulative time from the PDD, with respect to when that stream was first started at the stream handler's request.

PDD Values

The stream handler passes a pointer to the PDD on DDCMD_SETUP. This points to a value used by the PDD for setting the referenced time of the PDD. We do not always want the PDD to start its time at 0 every time the stream handler does a start, because the stream handler might have performed a seek in the stream. The PDD might have played a minute of data and then seeked backwards to maybe the 30-second mark in the data. If we issue a start, we do not want the PDD to think it is starting from zero again when it is really starting from the 30-second mark in that stream.

DDCMD_CONTROL has an important NOTIFY subfunction, which is used for cuepoint or event detection. The stream handler supports events in cuepoints-an application request to be notified when a particular location in the file is reached or a specific time period has elapsed. The stream handler uses two methods for detecting how much time has elapsed:

1. Using DDCMD_CONTROL NOTIFY, the stream handler requests to be notified by the PDD at a particular time and passes a pointer to the cue time.

2. The stream handler determines the time internally. This method is not as precise as the first method, because only the PDD knows the real time.

For example, suppose the stream handler does a DDCMD_CONTROL NOTIFY at one minute. If the PDD supports precise event detection, it must accept this request and put it into a queue somewhere, preferably a linked list. This linked list will have the time of one minute so that during the streaming process, the PDD occasionally checks to see whether it is at the one minute mark. When this event occurs, the PDD calls back on an SHD_REPORT_EVENT. Then, the programmer can free up that event detection linked list node.

Keep in mind that the PDD should have the capability to queue these requests because there may additional requests. For example, an application might request to be notified at the one-minute mark, next at a minute and a half, and possibly every five minutes.

If the PDD does not support event detection, then when it gets called on a DDCMD_CONTROL NOTIFY, he responds ERROR_INVALID_REQUEST. This response tells the stream handler that it must do the event detection itself.

Note: Refer to the *OS/2 Multimedia Programming Reference* for the return codes for the interfaces of the IDC.

Performance-Tuning the Sync/Stream Manager

The Sync/Stream Manager is comprised of two modules: SSMDD.SYS and SSM.DLL. SSMDD.SYS is the Ring 0 device driver of the Sync/Stream Manager. Its syntax is as follows:

```
DEVICE=                              SSMDD.SYS
           drive        path                      /S:sss
                                                  /P:ppp
                                                  /H:hhh
                                                  /Q:qqq
                                                  /E:eee
```

Parameters:

/S:*sss*	Specifies the number of streams that can be created at the same time. Values range from 1 through 64. The default value for machines with more than 8MB of memory is 12. The default value for machines with 8MB of memory or less is 6.
/P:*ppp*	Specifies the number of processes that can create streams at the same time. Values range from 1 through 64. The default value for machines with more than 8MB of memory is 12. The default value for machines with 8MB of memory or less is 6.
/H:*hhh*	Specifies the maximum amount of heap space (KB) that will be used. Values range from 16 through 256; the default value is 64.
/Q:*qqq*	Specifies the size of the event queue (per process). Values range from 2 through 1024; the default value is 64.
/E:*eee*	Specifies the number of events that can be enabled (per stream). Values range from 1 through 1024. The default value for machines with more than 8MB of memory is 32. The default value for machines with 8MB of memory or less is 20.

Note: The DEVICE=SSMDD.SYS statement must appear as the first Ring 0 stream handler statement in the CONFIG.SYS file.

Sync/Stream Resource Limits

- The maximum number of streams is 64.
- The maximum number of streams in a sync group is 64.
- The maximum number of processes controlling streams is 64.
- The maximum size of Sync/Stream Manager event queue per process is 1024 entries.

I/O Procedures

This section illustrates how to write a custom file format I/O procedure (IOProc). Source code is provided for the following I/O procedures located in the \TOOLKIT\SAMPLES\MM subdirectory:

Case-Converter
: Provides a simple example of how to write a file format I/O procedure (without illustrating the use of data translation). This sample performs case conversion of text. See the \TOOLKIT\SAMPLES\MM\CASECONV subdirectory.

M-Motion
: Provides an example of how to write an I/O procedure for use with image file formats. This sample enables file format

transparency for M-Motion still video files and illustrates the use of data translation. See the \TOOLKIT\SAMPLES\MM\MMIOPROC subdirectory.

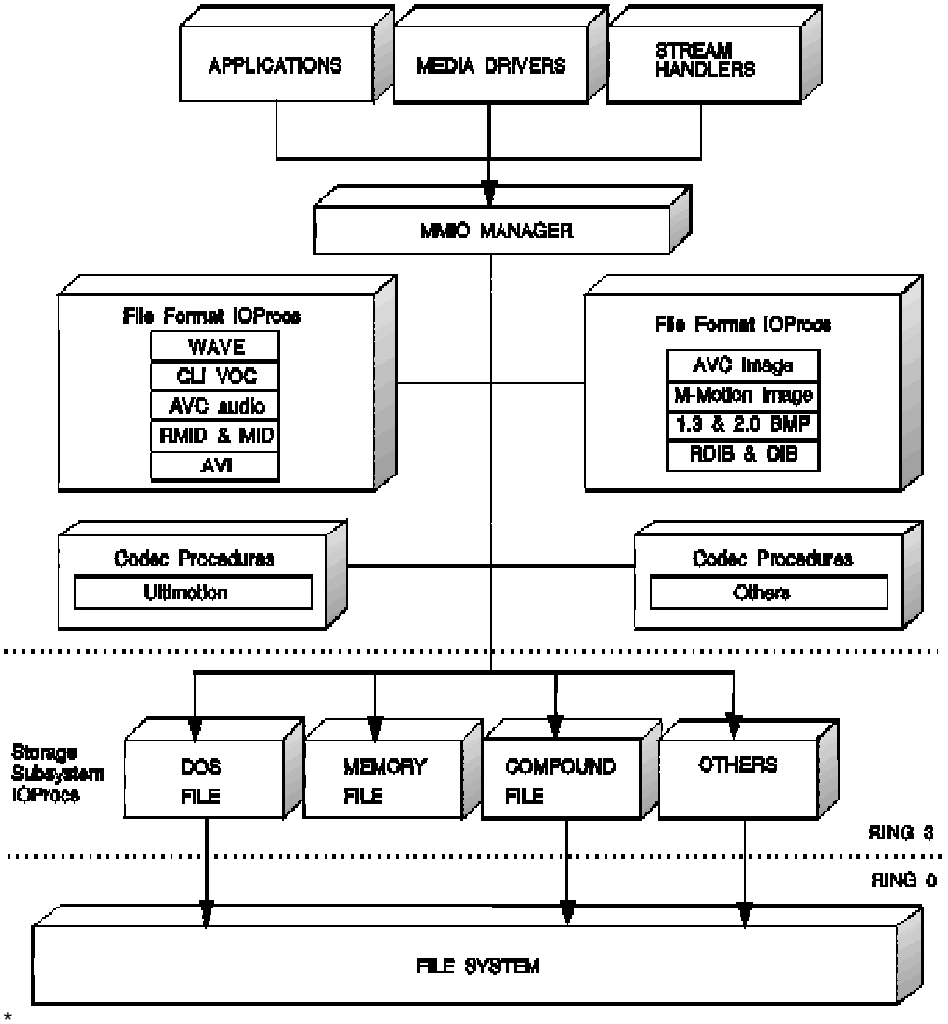
Ultimotion*

Provides a detailed example of what you need to consider when writing I/O procedures for software motion video file formats. ULIOT includes CODEC support and illustrates how to integrate common and file-format-specific code to support multiple I/O procedures. See the \TOOLKIT\SAMPLES\MM\ULTIMOIO subdirectory.

The following discussion focuses on the messages the M-Motion I/O procedure samples support, and what minimal processing is required for the messages. Information is also provided on the Ultimotion I/O procedure, which illustrates how to call and initialize a CODEC procedure.

I/O Procedure Architecture

The MMIO subsystem of OS/2 multimedia isolates applications, media control drivers, and stream handlers from data-specific processing in the same way that the media control interface buffers applications from device-specific processing. Applications send MMIO functions through the MMIO Manager, which uses I/O procedures (IOProcs) to manipulate specific types of multimedia data. The following figure illustrates the procedures available with the installation of OS/2 multimedia.



*

Message Handling

An IOProc is a message-based handler. Applications and the MMIO subsystem communicate to IOProcs through the use of MMIO messages. When MMIO receives a request from an application, the MMIO Manager sends a message for that operation to the IOProc that is responsible for that particular file format or storage system. In turn, the I/O procedure performs operations based on the messages it receives from the MMIO Manager or an application.

MMIO messages can be either pre-defined or user-defined messages:

Pre-defined Message
: This is a system message that is sent by the MMIO Manager for its associated function. For example, an application issuing an mmioOpen function causes the MMIO Manager to send an MMIOM_OPEN message to an I/O procedure to open a specific file. These messages enable applications to manage media files in a format-independent manner. The MMIO Manager determines the correct I/O procedure to process the message, based on an I/O procedure identifier and I/O procedure type specified in the function. See I/O Procedure Identifier (FOURCC) and I/O Procedure Type.

User-defined Message
: This is a private message sent directly to an I/O procedure from an application through the use of the mmioSendMessage function. This function enables a program to call an I/O procedure directly (unlike system messages, which are sent by the MMIO Manager). MMIOOS2.H in the \TOOLKIT\H subdirectory defines the identifier MMIO_USER so that you can create your own messages. The mmioSendMessage function requires that your custom messages be defined at or above the MMIOM_USER value defined in the MMIOOS2.H file.

I/O Procedure Identifier (FOURCC)

A FOURCC is a 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters (padded on the right with blank characters). Each I/O procedure supports a specific file format. The file format and IOProc are represented by a specific FOURCC code. This permits the FOURCC to be used as an ID value, rather than the National Language Support (NLS) string-name of the file format or a file name extension.

Formats that support multiple media types require a different FOURCC for each variation. This appears as a different I/O procedure for each media type.

Note: Use the mmioIdentifyFile function to identify the four-character code if it is not available.

I/O Procedure Type

Certain MMIO functions operate on a specific IOProc type. There are two types of I/O procedures: *file format* and *storage system* . A *file format* IOProc operates on the contents of a file, and calls other MMIO services when required. In contrast, a *storage system* IOProc operates on the storage presentation of the media object, and calls base operating system services.

To indicate an I/O procedure type during initialization, set the *ulIOProcType* field in the MMFORMATINFO structure to either MMIO_IOPROC_FILEFORMAT for file format IOProcs or MMIO_IOPROC_STORAGESYSTEM for storage system IOProcs. You should also have a resource file, which specifies the NLS name used to describe the I/O procedure. You must bind the RC file to the DLL if the I/O procedure is to be used in multiple countries. The IOProc needs to handle the GETFORMATINFO and GETFORMATNAME messages to provide the above information.
File Format I/O Procedure

A file format IOProc should support all MMIO system messages (with the exception of RIFF compound file messages). It should also handle any user-defined messages created by the application. For example, a file format IOProc needs to support the MMIOM_GETFORMATINFO message, because the MMIO Manager internally issues this message to an IOProc when it is being installed. If the MMIOM_GETFORMATINFO message is not supported, a blank MMFORMATINFO structure is placed on the MMIO internal IOProc table for that specific IOProc, except for the FOURCC.

In addition, system messages should be supported by a default message handler, which reports back to MMIO that the message is unsupported. This message handler should attempt to pass any message it cannot support to a subsequent child IOProc. For example, a message is passed from a file format IOProc to a storage system IOProc as shown in the following example.

```
default:
    {
     /*
      * Declare Local Variables.
      */
     PMMFILESTATUS       pVidInfo;
     LONG                lRC;
     /*********************************************************
      * Check for valid MMIOINFO block.
      *********************************************************/
     if (!pmmioinfo)
         return (MMIO_ERROR);
     /*********************************************************
      * Set up our working variable MMFILESTATUS.
      *********************************************************/
     pVidInfo = (PMMFILESTATUS) pmmioinfo->pExtraInfoStruct;

     if (pVidInfo != NULL && pVidInfo->hmmioSS)
        {
         lRC = mmioSendMessage (pVidInfo->hmmioSS,
                                usMsg,
                                lParam1,
                                lParam2);
         if (!lRC)
            pmmioinfo->ulErrorRet = mmioGetLastError (pVidInfo->hmmioSS);
         return (lRC);
        }
    else
       {
        if (pmmioinfo != NULL)
           pmmioinfo->ulErrorRet = MMIOERR_UNSUPPORTED_MESSAGE;
        return (MMIOERR_UNSUPPORTED_MESSAGE);
       }

    }   /* end case of Default */
```

If you write a custom IOProc that supports translation, the following messages need to consider the translate flags:

- MMIOM_OPEN
- MMIOM_READ
- MMIOM_WRITE
- MMIOM_SEEK
- MMIOM_CLOSE
- MMIOM_GETHEADER
- MMIOM_SETHEADER
- MMIOM_QUERYHEADER

Storage System I/O Procedure

A storage system IOProc typically handles a subset of the system-defined messages that operate on the storage system. For example, the DOS and MEM IOProcs handle the following messages:

- MMIOM_OPEN
- MMIOM_READ
- MMIOM_WRITE
- MMIOM_SEEK
- MMIOM_CLOSE
- MMIOM_GETFORMATNAME
- MMIOM_GETFORMATINFO
- MMIOM_IDENTIFYFILE

A message specific to the storage system IOProc, such as extended attributes of a file, would pass through the default handler of the file format IOProc to the storage system IOProc for processing.

Note: The RIFF compound file (CF) IOProc installed with OS/2 multimedia only supports MMIOM_IDENTIFYFILE, MMIOM_GETFORMATINFO, and MMIOM_GETFORMATNAME. The compound file (CF) IOProc does not need to support additional MMIO messages because the bundle (BND) IOProc performs the direct file I/O operations. These two IOProcs can be viewed as one logical compound file IOProc.

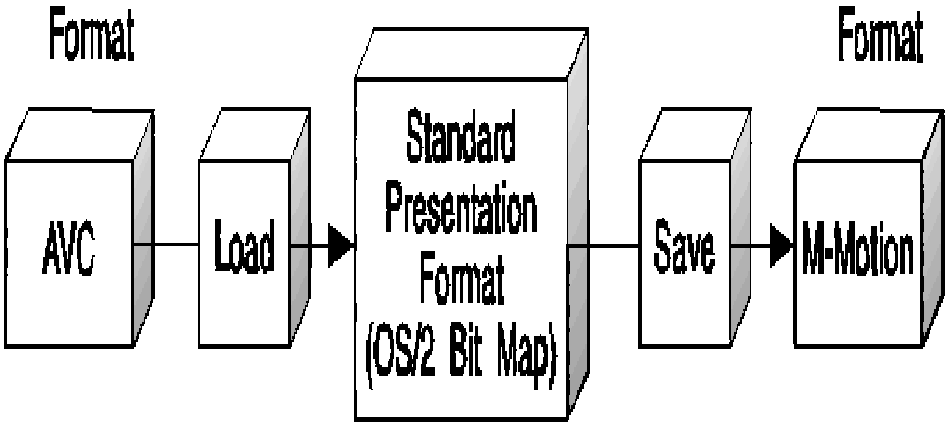
Data Translation and File Conversion

MMIO provides a set of options in its API to support two modes of file access-*translated* and *untranslated* . These modes enable an application to access data in its pure, proprietary format, or in a standard presentation format when performing its I/O operations. An IOProc can be written to optionally support both access methods.

The default mode of access, *untranslated* , allows the caller to perform I/O of file data in its native format. All header information and any data is written to a file or read from a file and presented at the caller level without modification.

The optional mode of access, *translated* , is the method used to mask proprietary data formats and allow a caller to use standardized header and data formats for a specific media type (for example, audio, image, or MIDI). A set of standard header formats and accompanying data formats have been defined as the standard presentation format for these purposes. An IOProc can be written to optionally support the standard format. It performs the translation of header information, data information, or both from its native format to the standard format for that media type during read and write type operations. The translation is performed for the file header and data.

With full translation capability enabled in pairs of IOProcs, it is possible to convert files from one format to another very easily. File conversion is simply a combination of loading from one file and saving to another. For example, an application can read from an AVC image file with translation enabled, to the standard presentation format, which is the OS/2 bitmap. The application can then use the bitmap as desired, including displaying it on the screen, image manipulating, and printing. Alternatively, the bitmap could be saved in a different file format such as M-Motion, by writing to the M-Motion IOProc with translation enabled. For example:



I/O procedures and applications must use the same standard presentation format of data defined for each media type for conversion to be enabled. (The media types defined are image, audio, MIDI, digital video, and movie.) These standard formats apply to media descriptive (header) information and media content. The standard *description structures* are supersets of the headers each file format normally uses. This permits all formats to place the subset of their information into the standard form for other applications to access. Similarly, each specific format can retrieve only the subset that is necessary for its purpose. The standard *content format* is a usable data format representation that maintains as much quality as possible.

Translation functions assist these standard forms in helping to ensure that data is portable between applications, IOProcs, and the operating system services. The structures containing descriptive information have fields that can be mapped to system structures, such as the OS/2 operating system's BITMAPINFOHEADER. The content format must be directly usable by the operating system and services, or by standard hardware devices.

The descriptive header and content formats are tightly coupled. If a file contains a media item, an application can query the header describing the media. The IOProc returns the header, which includes the supported content format most closely matching the information actually in the file. For example, if an image file contains 21-bit YUV data, the IOProc for that file informs the application that it is providing 24-bit RGB. The IOProc is responsible for translating all subsequent read operations from YUV to RGB. In addition, when an application is creating a new media element, it can set the header for a new media item. All subsequent translated write operations, which are sent from the application to the IOProc, must contain data in the content format described by the header.

Each data type uses different description structures and content formats. The following table gives an overview of the standard presentation formats for supported media types.

Media	Header	Data
Audio	MMAUDIO	PCM 11.025, 22.05, 33.1 Khz
Image	MMIMAGE	OS/2 1.3 bitmap (24 bit RGB, 1, 4, 8 bit palette)

MIDI	MMMIDI	Format 0 or 1
MOVIES	MMMOVIE	Multi-track video and audio
VIDEO	MMVIDEO	16, 24 bit RGB, 4, 8 bit palette

Note: Data translation in compound multimedia files is only performed on media elements in the file. Translation is not performed on non-multimedia files.

MMFORMATINFO Data Structure

Several MMIO functions use the MMFORMATINFO data structure for media conversions. The mmioOpen function includes MMIO_TRANSLATEHEADER and MMIO_TRANSLATEDATA flags which are defined in the *ulTranslate* field of the MMIOINFO structure. All subsequent read and write operations of multimedia files return data based on these flags. Translation is currently defined only for image and audio. The MMIOOS2.H header file defines the MMFORMATINFO structure as shown in the following example.

```
typedef struct _MMFORMATINFO {         /* mmformatinfo                */
                                       */
   ULONG    ulStructLen;               /* Length of this structure    */
   FOURCC   fccIOProc;                 /* IOProc identifier           */
   ULONG    ulIOProcType;              /* Type of IOProc              */
   ULONG    ulMediaType;               /* Media type                  */
   ULONG    ulFlags;                   /* IOProc capability flags     */
   CHAR     szDefaultFormatExt[sizeof(FOURCC) + 1];  */
                                       /* Default extension 4 + null  */
   ULONG    ulCodePage;                /* Code page                   */
   ULONG    ulLanguage;                /* Language                    */
   LONG     lNameLength;               /* Length of identifier string */
   } MMFORMATINFO;
```

I/O Procedure Entry Point

The following example illustrates the the entry point used to access the functionality of an I/O procedure.

```
LONG APIENTRY IOProc_Entry ( PVOID   pmmioStr,
                             USHORT  usMessage,
                             LONG    lParam1,
                             LONG    lParam2)
```

Associated parameters include the following.

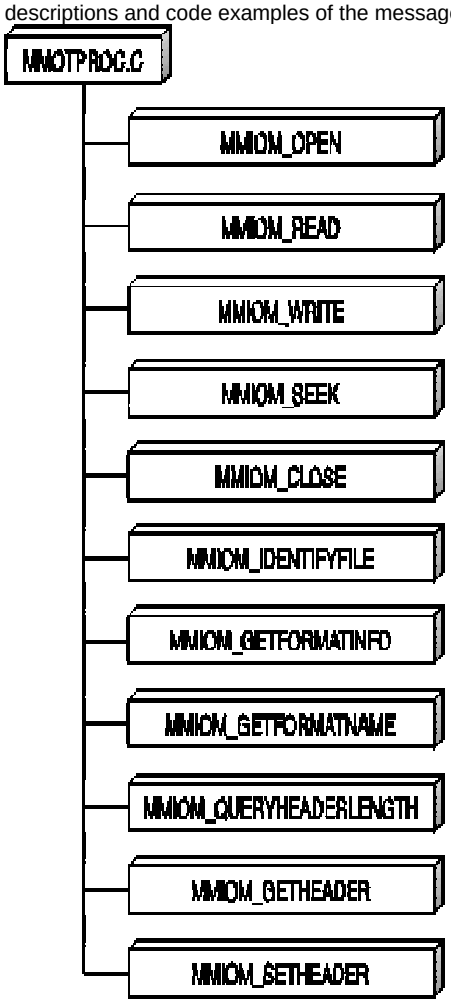
Parameter	Description
PVOID *pmmioStr*	Specifies a pointer to an MMIOINFO data structure that contains information about the open file.
USHORT *usMsg*	Specifies the message that the file I/O procedure is being asked to process. (User-defined messages must have messages defined above MMIOM_USER.)
LONG *lParam1*	Specifies message-dependent information such

```
                as a file name.

LONG lParam2    Specifies additional message-dependent
                information.  (Used with some messages as
                values.)
```

Note: The return value is message-dependent. If the I/O procedure does not recognize a message passed in by *usMsg* , and the default message handler does not recognize *usMsg* , then it must return MMIOERR_UNSUPPORTED_MESSAGE.

Supported Messages

The following figure illustrates the messages supported by the M-Motion I/O procedure (MMOTTK.DLL). Following the figure are descriptions and code examples of the messages file format I/O procedures must support.



MMIOM_OPEN

Each IOProc must be able to process the MMIOM_OPEN message, which requests that a file be opened. Once the application knows which

IOProc is associated with the selected file, it can open the file using mmioOpen. The application references the appropriate IOProc using the FOURCC provided by the identification process.

A file format IOProc must check for the following items when the MMIO Manager issues an MMIOM_OPEN message.

- File format IOProcs use the *fccChildIOProc* field from the *pmmioinfo* structure and perform another mmioOpen. The MMIO_NOIDENTIFY flag must be set in this case.
- The *lLogicalFilePos* of the MMIOINFO structure should be set to either 0 or at the first byte of data following the header, if any. This example has a header and *lLogicalFilePos* is set using the return code from mmioSeek.
- A file format IOProc must check to see if the MMIO_TRANSLATEDATA or MMIO_TRANSLATEHEADER flag is set. If a translate flag is set, it processes the data according to a set of defined interchange formats (refer to the *OS/2 Multimedia Programming Reference* for details). If a translate flag is not set, it allows the data to pass through the IOProc with application-specific modifications. Translation support is required if the IOProc is to be supported under the Multimedia Data Converter program.

If the OPEN was successful, the application can obtain information about the media in the file using the mmioGetHeaderInfo message.

The following example illustrates how to handle the MMIOM_OPEN message for a file format IOProc. The MMIOM_OPEN message handler uses the mmioOpen function to locate a media data object using an MMIO-supported storage system IOProc. Upon opening the data object, an *hmmio* handle (H1) is returned to the file format IOProc. This handle is saved in the *aulInfo* [1] field of MMIOINFO for the file format IOProc. Upon the return to the mmioOpen function issued by the application, you will notice that another handle (H2) was already generated and returned to the application. These handles allow access to the data object. The application will use H2, and the file format IOProc will use H1 with MMIO function calls to the storage system IOProc.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_OPEN message.

```
case MMIOM_OPEN:
    {
    /*********************************************************
     * Declare local variables
     *********************************************************/
    PMMFILESTATUS   pVidInfo;   /* pointer to an M-Motion file    */
                                /* status structure that we will*/
                                /* use for this file instance   */
    MMIMAGEHEADER   MMImgHdr;
    ULONG           ulRequiredFileLength;
    ULONG           ulActualFileLength;
    ULONG           ulWidth;
    ULONG           ulHeight;
    PBYTE           lpYUVBuf;
    ULONG           ulRowCount;
    ULONG           ulRGBBytesPerLine;
    ULONG           ulYUVBytesPerLine;
    LONG            rc;
    HMMIO           hmmioSS;
    PBYTE           lpRGBBufPtr;
    FOURCC          fccStorageSystem;   /* SS I/O Proc FOURCC       */
    MMIOINFO        mmioinfoSS;         /* I/O info block for SS ref*/
    PSZ pszFileName = (CHAR *)lParam1;  /* get the filename from    */
                                        /* parameter                */
    /*********************************************************
     * Check for valid MMIOINFO block.
     *********************************************************/
    if (!pmmioinfo)
        return (MMIO_ERROR);
    /*********************************************************
     * If flags show read and write then send back an error.  We
     * only support reading or writing but not both at the same
     * time on the same file.
     *********************************************************/
    if ((pmmioinfo->ulFlags & MMIO_READWRITE) &&
        ((pmmioinfo->ulTranslate & MMIO_TRANSLATEDATA) ||
         (pmmioinfo->ulTranslate & MMIO_TRANSLATEHEADER)))
        {
        return (MMIO_ERROR);
        }
    /*****************************************************
     *   Determine the storage system/child IOProc that actually
     *   obtains the data for us.  The M-Motion data may be contained
     *   in a memory (RAM) file, as a component in a database or
     *   library (a Compound file), or as a stand-alone disk file.
     *
     *   While the application uses this M-Motion IOProc to obtain
     *   untranslated (M-Motion) or translated (bitmap) data,
```

```
 *   the IOProc must obtain it's data from something that
 *   reads and writes to a storage media.  The exact storage
 *   media is immaterial - so long as the read and write
 *   operations generate data that LOOKS like it is part
 *   of a standard file.
 ************************************************************/
if (!pmmioinfo->fccChildIOProc)
   {
   /* Need to determine SS if create from pmmioinfo and filename. */
   if (pmmioinfo->ulFlags & MMIO_CREATE)
      {
      if (mmioDetermineSSIOProc( pszFileName,
                                 pmmioinfo,
                                 &fccStorageSystem,
                                 NULL ))
         {
         fccStorageSystem = FOURCC_DOS;
         }
      }
   else
      {
      if (mmioIdentifyStorageSystem( pszFileName,
                                     pmmioinfo,
                                     &fccStorageSystem ))
         {
         return (MMIO_ERROR);
         }

      }

   if (!fccStorageSystem)
      {
      return (MMIO_ERROR);
      }
   else
      {
      pmmioinfo->fccChildIOProc = fccStorageSystem;
      }
   } /* end storage system identification block */
/************************************************************
 * Direct the open to the specific storage system necessary
 ************************************************************/
memset( &mmioinfoSS, '\0', sizeof(MMIOINFO));
memmove( &mmioinfoSS, pmmioinfo, sizeof(MMIOINFO));
mmioinfoSS.pIOProc = NULL;
mmioinfoSS.fccIOProc = pmmioinfo->fccChildIOProc;
mmioinfoSS.ulFlags |= MMIO_NOIDENTIFY;
/************************************************************
 * Try to open the file.  Add the NO IDENTIFY flag to
 *    ENSURE THAT WE DON'T LOOP RECURSIVELY!!!
 ************************************************************/
hmmioSS = mmioOpen (pszFileName,
                    &mmioinfoSS,
                    mmioinfoSS.ulFlags);
/************************************************************
 * Check if a DELETE was requested - mmioOpen returns a 1,
 *    so we much check this separately
 ************************************************************/
if (pmmioinfo->ulFlags & MMIO_DELETE)
   {
   /* was the delete successful?         */
   if (!hmmioSS)
      {
      pmmioinfo->ulErrorRet = MMIOERR_DELETE_FAILED;
      return (MMIO_ERROR);
      }
   else
      {
      return (MMIO_SUCCESS);
      }
   }
/************************************************************
 * Check the return code from the open call for an error.
 *    If not delete, then the open should have worked.
 ************************************************************/
if (!hmmioSS)
    return (MMIO_ERROR);
/************************************************************
 * Allocate memory for one M-Motion FileStatus structures
```

```
 ************************************************************/
DosAllocMem ((PPVOID) &pVidInfo,
                      sizeof (MMFILESTATUS),
                      fALLOC);
/************************************************************
 * Ensure the allocate was successful.  If not, then
 *   close the file and return open as unsuccessful...
 ************************************************************/
if (!pVidInfo)
   {
   mmioClose (hmmioSS, 0);
   return (MMIO_ERROR);
pVidInfo->hmmioSS = hmmioSS;
/************************************************************
 * Store pointer to our MMFILESTATUS structure in
 * pExtraInfoStruct field that is provided for our use.
 ************************************************************/
pmmioinfo->pExtraInfoStruct = (PVOID)pVidInfo;
/************************************************************
 * Set the fields of the FileStatus structure that the
 * IOProc is responsible for.
 ************************************************************/
InitFileStruct (pVidInfo);
/*******************************************************
 * If this is a read, we need to check that is a M-Motion
 *    file and perhaps get the data.
 *******************************************************/
if (pmmioinfo->ulFlags & MMIO_READ)
   {
   /*******************************************************
    * First we must get some basic information from the file
    * Read in data to fill up the MMOTIONHEADER structure.
    *
    * If the read is unsuccessful, this is not a M-Motion file
    * and we should return a failure on the open
    *******************************************************/
   if (sizeof (MMOTIONHEADER) !=
                       mmioRead (pVidInfo->hmmioSS,
                                 (PVOID) &pVidInfo->mmotHeader,
                                 (ULONG) sizeof (MMOTIONHEADER)))
       {
       mmioClose (pVidInfo->hmmioSS, 0);
       DosFreeMem ((PVOID) pVidInfo);
       return (MMIO_ERROR);
       }
   /* Ensure this IS an M-Motion file header before we continue  */
   if (strcmp (pVidInfo->mmotHeader.mmID, "YUV12C"))
       {
       mmioClose (pVidInfo->hmmioSS, 0);
       DosFreeMem ((PVOID) pVidInfo);
       return (MMIO_ERROR);
       }
   /*******************************************************
    * Set up width and height of image.
    *******************************************************/
   ulWidth  = (ULONG)pVidInfo->mmotHeader.mmXlen;
   ulHeight = (ULONG)pVidInfo->mmotHeader.mmYlen;
   /* Calculate what the length of the file SHOULD be based on the */
   /*   header contents                                             */
   ulRequiredFileLength = (((ulWidth >> 2) * 6) * ulHeight) +
                                        sizeof (MMOTIONHEADER);

   /* Query what the ACTUAL length of the file is,                 */
   /*   then move back to just after the header.                   */
   ulActualFileLength = (ULONG)mmioSeek (pVidInfo->hmmioSS,
                                         0, SEEK_END);

   mmioSeek (pVidInfo->hmmioSS, sizeof (MMOTIONHEADER), SEEK_SET);
   /* If these don't match, then it isn't a VALID M-Motion file    */
   /*    - regardless of what the header says.                     */
   if (ulRequiredFileLength != ulActualFileLength)
       {
       mmioClose (pVidInfo->hmmioSS, 0);
       DosFreeMem ((PVOID) pVidInfo);
       return (MMIO_ERROR);
       }

   /*******************************************************
    * If the app intends to read in translation mode, we must
```

```
 * allocate and set-up the buffer that will contain the RGB data.
 *
 * We must also read in the data to insure that the first
 * read, seek, or get-header operation will have data
 * to use.  This is ONLY NECESSARY FOR TRANSLATED MODE
 * operations, since we must process reads/writes pretending
 * the image is stored from the bottom-up.
 *
 ************************************************************
 ************************************************************
 * Fill out the MMIMAGEHEADER structure.
 ************************************************************/
MMImgHdr.ulHeaderLength = sizeof (MMIMAGEHEADER);
MMImgHdr.ulContentType  = MMIO_IMAGE_PHOTO;
MMImgHdr.ulMediaType    = MMIO_MEDIATYPE_IMAGE;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cbFix =
            sizeof (BITMAPINFOHEADER2);
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cx              = ulWidth;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cy              = ulHeight;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cPlanes         = 1;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cBitCount       = 24;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.ulCompression   =
    BCA_UNCOMP;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cbImage         =
    ulWidth * ulHeight * 3;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cxResolution    = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cyResolution    = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cclrUsed        = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cclrImportant   = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.usUnits         = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.usReserved      = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.usRecording     =
    BRA_BOTTOMUP;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.usRendering     =
    BRH_NOTHALFTONED;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cSize1          = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.cSize2          = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.ulColorEncoding = 0L;
MMImgHdr.mmXDIBHeader.BMPInfoHeader2.ulIdentifier    = 0L;
/************************************************************
 * Copy the image header into private area for later use.
 * This will be returned on a mmioGetHeader () call
 ************************************************************/
pVidInfo->mmImgHdr = MMImgHdr;
/*******************************************************
 * YUV Bytes/Line are = 1 1/2 times the number of pels
 *******************************************************/
ulYUVBytesPerLine = ulWidth + (ulWidth >> 1);

/*******************************************************
 * RGB Bytes/Line are = 2* YUV bytes/line
 *******************************************************/
ulRGBBytesPerLine = (ulYUVBytesPerLine << 1);

/*******************************************************
 * Determine total bytes in image
 *******************************************************/
pVidInfo->ulRGBTotalBytes = ulWidth * ulHeight * 3;

/*******************************************************
 * M-Motion images are always on 4-pel boundaries, which also
 * makes them on 4-byte/LONG boundaries, which is used for
 * bitmaps.  Therefore, there are no extra pad bytes necessary.
 *******************************************************/
pVidInfo->ulImgPaddedBytesPerLine = ulWidth * 3;
pVidInfo->ulImgTotalBytes = pVidInfo->ulRGBTotalBytes;
/*******************************************************
 * For translated data READ mode, we must allocate a buffer,
 * get the YUV data from the file, and load the RGB buffer.
 * Place format-specific code here to load the image into the
 * buffer.  The code below is M-Motion format specific.
 *******************************************************/
if (pmmioinfo->ulTranslate & MMIO_TRANSLATEDATA)
   {
   /*****************************************************
    * Get space for full image buffer.
    * This will be retained until the file is closed.
    *****************************************************/
   if (DosAllocMem ((PPVOID) &(pVidInfo->lpRGBBuf),
```

```
                    pVidInfo->ulRGBTotalBytes,
                    fALLOC))
        {
        mmioClose (pVidInfo->hmmioSS, 0);
        DosFreeMem ((PVOID) pVidInfo);
        return (MMIO_ERROR);
        }

      /*****************************************************
       * Get temporary space for one line YUV buffer.
       *****************************************************/
      if (DosAllocMem ((PPVOID) &lpYUVBuf,
                     ulYUVBytesPerLine,
                     fALLOC))
        {
        mmioClose (pVidInfo->hmmioSS, 0);
        DosFreeMem ((PVOID) pVidInfo);
        return (MMIO_ERROR);
        }
      /*****************************************************
       * Initialize the beginning buffer position.
       *****************************************************/
      lpRGBBufPtr = pVidInfo->lpRGBBuf;

      /*****************************************************
       * Read in YUV data one line at a time, converting
       * from YUV to RGB, then placing in the image buffer.
       *****************************************************/
      for (ulRowCount = 0;
           ulRowCount < ulHeight;
           ulRowCount++)

        {
        /*************************************************
         * Read in one line.
         *************************************************/
        rc = mmioRead (pVidInfo->hmmioSS,
                       (PVOID) lpYUVBuf,
                       (ULONG) ulYUVBytesPerLine);

        /*************************************************
         * Convert one line at a time.
         *************************************************/
        ConvertOneLineYUVtoRGB (lpYUVBuf,
                                lpRGBBufPtr,
                                ulYUVBytesPerLine);

        /*************************************************
         * Make sure buffer ptr is correct for the next convert.
         *************************************************/
        lpRGBBufPtr += (LONG)ulRGBBytesPerLine;
        }   /* end of FOR loop to read YUV data */
      DosFreeMem (lpYUVBuf);

      /*****************************************************
       * This changes from M-Motion's top-down form to OS/2
       * PM's bottom-up bitmap form.
       *****************************************************/
      ImgBufferFlip (pVidInfo->lpRGBBuf,
                     pVidInfo->ulImgPaddedBytesPerLine,
                     ulHeight);
      /*****************************************************
       * RGB buffer now full, set position pointers to the
       * beginning of the buffer.
       *****************************************************/
      pVidInfo->lImgBytePos =  0;
      }   /*  end IF TRANSLATED block */
   } /* end IF READ block */
return (MMIO_SUCCESS);
} /* end case of MMIOM_OPEN */
```

MMIOM_READ and MMIOM_WRITE

The MMIOM_READ message requests that bytes be read from an open file; MMIOM_WRITE message requests that bytes be written to an open file. These messages should be handled differently for each IOProc, depending on the requirements imposed by the file's data. Because a file might be using buffered I/O, mmioRead and mmioWrite maintain the *lBufOffset* and the *lDiskOffset* fields. The IOProc should not modify these fields. If these fields are needed by the IOProc, the IOProc can use the *aulInfo* array to maintain. Additionally, the *pExtraInfoStruct* can be used for any user-defined structure that the IOProc requires. The sample IOProc stores its header in this field to demonstrate this capability. If the IOProc is a file format IOProc, it should use mmioRead or mmioWrite calls to the storage system IOProc, using the internal handle generated during the open processing. A storage system IOProc might simply issue calls to DosRead or DosWrite.

To implement a file format IOProc for translation mode, and provide support for the MMIO_TRANSLATEDATA flag, additional code is required for the MMIOM_READ and MMIOM_WRITE message processing. During read processing, after data is read from the file to a private buffer in its native encoding format, the data must be translated from its native encoding scheme to the standard presentation format encoding scheme for its media type. The translated data is then presented to the application in its read buffer. Likewise, for write processing, data is received from the application in the standard presentation format, and must be translated to its native encoding scheme before being written to the file.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_READ and MMIOM_WRITE messages.

```
case MMIOM_READ:
     {
     /*********************************************************
      * Declare Local Variables
      *********************************************************/
     PMMFILESTATUS   pVidInfo;
     LONG            rc;
     LONG            lBytesToRead;

     /*********************************************************
      * Check for valid MMIOINFO block.
      *********************************************************/
     if (!pmmioinfo)
         return (MMIO_ERROR);

     /*********************************************************
      * Set up our working file status variable.
      *********************************************************/
     pVidInfo = (PMMFILESTATUS)pmmioinfo->pExtraInfoStruct;

     /*********************************************************
      * Is Translate Data off?
      *********************************************************/
     if (!(pmmioinfo->ulTranslate & MMIO_TRANSLATEDATA))
         {
         /*****************************************************
          * Since no translation, provide exact number of bytes req.
          *****************************************************/
         if (!lParam1)
             return (MMIO_ERROR);

         rc = mmioRead (pVidInfo->hmmioSS,
                        (PVOID) lParam1,
                        (ULONG) lParam2);

         return (rc);
         }

     /*********************************************************
      * Otherwise, Translate Data is on...
      *********************************************************/

     /*********************************************************
      * Ensure we do NOT write more data out than is remaining
      *    in the buffer.  The length of read was requested in
      *    image bytes, so confirm that there are that many of
      *    virtual bytes remaining.
      *********************************************************/
     if ((ULONG)(pVidInfo->lImgBytePos + lParam2) >
          pVidInfo->ulImgTotalBytes)
        lBytesToRead =
            pVidInfo->ulImgTotalBytes - pVidInfo->lImgBytePos;
     else
        lBytesToRead = (ULONG)lParam2;

     /*********************************************************
      * Perform this block on ALL reads.  The image data should
```

```
     * be in the RGB buffer at this point, and can be handed
     * to the application.
     *
     * Conveniently, the virtual image position is the same
     *    as the RGB buffer position, since both are 24 bit-RGB
     ************************************************************/
    memcpy ((PVOID)lParam1,
            &(pVidInfo->lpRGBBuf[pVidInfo->lImgBytePos]),
            lBytesToRead);

    /************************************************************
     * Move RGB buffer pointer forward by number of bytes read.
     * The Img buffer pos is identical since both are 24 bits.
     ************************************************************/
    pVidInfo->lImgBytePos += lBytesToRead;

    return (lBytesToRead);
    }   /* end case  of MMIOM_READ */

case MMIOM_WRITE:
     {
     /************************************************************
      * Declare Local Variables.
      ************************************************************/
     PMMFILESTATUS       pVidInfo;
     USHORT              usBitCount;
     LONG                lBytesWritten;
     ULONG               ulImgBytesToWrite;

     /************************************************************
      * Check for valid MMIOINFO block.
      ************************************************************/
     if (!pmmioinfo)
         return (MMIO_ERROR);

     /************************************************************
      * Set up our working variable MMFILESTATUS.
      ************************************************************/
     pVidInfo = (PMMFILESTATUS) pmmioinfo->pExtraInfoStruct;

     /************************************************************
      * See if a SetHeader has been done on this file.
      ************************************************************/
     if ((!pVidInfo) || (!pVidInfo->bSetHeader))
         {
         return (MMIO_ERROR);
         }

     if (!(pmmioinfo->ulTranslate & MMIO_TRANSLATEDATA))
         {
         /*******************************************************
          * Translation is off, take amount of bytes sent and
          * write to the file.
          *******************************************************
          * Ensure that there is a data buffer to write from.
          *******************************************************/
         if (!lParam1)
             return (MMIO_ERROR);

         lBytesWritten = mmioWrite (pVidInfo->hmmioSS,
                                    (PVOID) lParam1,
                                    (ULONG) lParam2);

         return (lBytesWritten);
         }

     /************************************************************
      * Translation is on.
      ************************************************************
      * Set up local variables.
      ************************************************************/
     usBitCount =
        pVidInfo->mmImgHdr.mmXDIBHeader.BMPInfoHeader2.cBitCount;


     /************************************************************
      * Ensure we do not attempt to write past the end of the
      *    buffer...
      ************************************************************/
```

```
      if ((ULONG)(pVidInfo->lImgBytePos + lParam2) >
         pVidInfo->ulImgTotalBytes)
         ulImgBytesToWrite =
             pVidInfo->ulImgTotalBytes - pVidInfo->lImgBytePos;
      else
         ulImgBytesToWrite = (ULONG)lParam2;

      /*********************************************************
       * Write the data into the image buffer.  It will be converted to
       *   RGB, then YUV  when the file is closed.  This allows the
       *   application to seek to arbitrary  positions within the
       *   image in terms of the bits/pel, etc they are writing.
       *********************************************************/
      memcpy (&(pVidInfo->lpImgBuf[pVidInfo->lImgBytePos]),
              (PVOID)lParam1,
              ulImgBytesToWrite);

      /* Update current position in the image buffer */
      pVidInfo->lImgBytePos += ulImgBytesToWrite;

      return (ulImgBytesToWrite);
      }   /* end case of MMIOM_WRITE */

   /*
    * If the IOProc has a child IOProc, then pass the message
    *  on to the Child, otherwise return Unsupported Message
    */
   default:
       {
        /*
         * Declare Local Variables.
         */
        PMMFILESTATUS       pVidInfo;
        LONG                lRC;

        /*********************************************************
         * Check for valid MMIOINFO block.
         *********************************************************/
        if (!pmmioinfo)
            return (MMIO_ERROR);

        /*********************************************************
         * Set up our working variable MMFILESTATUS.
         *********************************************************/
        pVidInfo = (PMMFILESTATUS) pmmioinfo->pExtraInfoStruct;

        if (pVidInfo != NULL && pVidInfo->hmmioSS)
           {
            lRC = mmioSendMessage (pVidInfo->hmmioSS,
                                   usMsg,
                                   lParam1,
                                   lParam2);
            if (!lRC)
               pmmioinfo->ulErrorRet = mmioGetLastError (pVidInfo->hmmioSS);
            return (lRC);
           }
       else
           {
            if (pmmioinfo != NULL)
               pmmioinfo->ulErrorRet = MMIOERR_UNSUPPORTED_MESSAGE;
            return (MMIOERR_UNSUPPORTED_MESSAGE);
           }

       }   /* end case of Default */

   } /* end SWITCH statement for MMIO messages */

return (0);
}      /* end of window procedure */
```

MMIOM_SEEK

This message is handled differently by each IOProc. Because of the requirements of different file formats, you will need to determine if this message can be supported, and if so, how to implement the three different types of seeks. A file format IOProc can use some type of calculation to determine the seek distance, and what those distance units are. The file format IOProc should use mmioSeek to call the storage system IOProc. A storage system IOProc can call the DOSSetFilePtr function to set the file's position. The bounds checking that must occur for compound file elements is handled by the mmioSeek function for BND file elements. Any other compound file IOProcs of this nature need to implement this checking in its own code.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_SEEK message.

```
case MMIOM_SEEK:
     {
     /*********************************************************
      * Set up locals.
      *********************************************************/
     PMMFILESTATUS   pVidInfo;
     LONG            lNewFilePosition;
     LONG            lPosDesired;
     SHORT           sSeekMode;

     /*********************************************************
      * Check to make sure MMIOINFO block is valid.
      *********************************************************/
     if (!pmmioinfo)
         return (MMIO_ERROR);

     /*********************************************************
      * Set up our working file status variable.
      *********************************************************/
     pVidInfo = (PMMFILESTATUS)pmmioinfo->pExtraInfoStruct;

     lPosDesired = lParam1;
     sSeekMode = (SHORT)lParam2;

     /*********************************************************
      * Is Translate Data on?
      *********************************************************/
     if (pmmioinfo->ulTranslate & MMIO_TRANSLATEDATA)
         {
         /*****************************************************
          * Attempt to move the Image buffer pointer to the
          * desired location.  App sends SEEK requests in
          * positions relative to the image planes & bits/pel
          * We must also convert this to RGB positions
          *****************************************************/
         switch (sSeekMode)
             {
             case SEEK_SET:
                 {
                 lNewFilePosition = lPosDesired;
                 break;
                 }
             case SEEK_CUR:
                 {
                 lNewFilePosition = pVidInfo->lImgBytePos + lPosDesired;
                 break;
                 }
             case SEEK_END:
                 {

                 lNewFilePosition =
                             pVidInfo->ulImgTotalBytes += lPosDesired;
                 break;
                 }

             default :
                return (MMIO_ERROR);
             }

         /*****************************************************
          * Make sure seek did not go before start of file.
          * If so, then don't change anything, just return an error
          *****************************************************/
         if (lNewFilePosition < 0)
            {
            return (MMIO_ERROR);
            }
```

```
        /*******************************************************
         * Make sure seek did not go past the end of file.
         *******************************************************/
        if (lNewFilePosition > (LONG)pVidInfo->ulImgTotalBytes)
            lNewFilePosition = pVidInfo->ulImgTotalBytes;

        pVidInfo->lImgBytePos = lNewFilePosition;

        return (pVidInfo->lImgBytePos);
        } /* end  IF DATA TRANSLATED */

     /*********************************************************
      * Translate Data is OFF...
      *********************************************************
      * if this is a seek from the beginning of the file,
      *    we must account for and pass the header
      *********************************************************/
     if (lParam2==SEEK_SET)
        lPosDesired += MMOTION_HEADER_SIZE;

     lNewFilePosition = mmioSeek (pVidInfo->hmmioSS,
                                  lPosDesired,
                                  sSeekMode);

     /*****************************************
      * Ensure we did not move to within the header
      *****************************************/
     if ((lNewFilePosition != MMIO_ERROR) &&
         (lNewFilePosition < MMOTION_HEADER_SIZE))
        {
        lNewFilePosition = mmioSeek (pVidInfo->hmmioSS,
                                     (LONG)MMOTION_HEADER_SIZE,
                                     SEEK_SET);
        }

     /*********************************************************
      * Return new position.  Always remove the length of the
      *    header from the this position value
      *********************************************************/
     if (lNewFilePosition != MMIO_ERROR)
        lNewFilePosition -= MMOTION_HEADER_SIZE;

     return (lNewFilePosition);
     }  /* end case of MMIOM_SEEK */
```

MMIOM_CLOSE

This message must be implemented by file format IOProcs to be able to properly close the file. Note that the mmioClose function will call the mmioFlush function to empty the file I/O buffers (so this does not need to be performed by the IOProc message handler).

A file format IOProc will call mmioClose with its internal HMMIO handle to close the file, unless it chooses to recognize the MMIO_FHOPEN flag available on mmioClose. This flag allows the close processing to occur, but still allow the file handle to remain open. If the file is an element of a compound file, it will need to possibly update the compound file header to reflect any changes that have been made to the file.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_CLOSE message.

```
case MMIOM_CLOSE:
    {
    /*******************************************************
     * Declare local variables.
     *******************************************************/
    PMMFILESTATUS   pVidInfo;         /* MMotion IOProc instance data */

    ULONG           ulHeight;         /* Image height               */
    USHORT          usBitCount;
    /* Image width, including overflow in 1bpp & 4bpp               */
    ULONG           ulImgPelWidth;
    PBYTE           lpYUVLine;        /* One line of packed YUV     */
```

```
LONG            lYUVBytesPerLine;

ULONG           ulMaxPelWidth;    /* # pels on 4-pel boundaries  */
/* # pels on a YUV line in the output file                      */
ULONG           ulYUVPelWidth;
ULONG           ulRGBMaxBytesPerLine; /* #bytes on 4-pel bounds  */
PBYTE           lpRGBLine;        /* One line of 24bit RGB       */

PBYTE           lpImgBufPtr;      /* Current loc in RGB image buf*/
LONG            lRGBBytesPerLine; /* #bytes on a line in image   */
ULONG           ulRowCount;       /* loop counter                */
LONG            lBytesWritten;    /* #bytes output on a write    */
LONG            lRetCode;
USHORT          rc;

/*********************************************************
 * Check for valid MMIOINFO block.
 *********************************************************/
if (!pmmioinfo)
    return (MMIO_ERROR);

/*********************************************************
 * Set up our working file status variable.
 *********************************************************/
pVidInfo = (PMMFILESTATUS)pmmioinfo->pExtraInfoStruct;

/*********************************************************
 * Assume success for the moment....
 *********************************************************/
lRetCode = MMIO_SUCCESS;

/*********************************************************
 * see if we are in Write mode and have a buffer to write out.
 *    We have no image buffer in UNTRANSLATED mode.
 *********************************************************/
if ((pmmioinfo->ulFlags & MMIO_WRITE) && (pVidInfo->lpImgBuf))
    {
    /*****************************************************
     * The buffer should be in palettized or 24-bit RGB
     * We must convert it to YUV to be written to the file.
     *****************************************************
     * The buffer should be complete.  If not, then we
     * should still close, but can flag an error to the
     * user that the data may be corrupted.  The only way
     * we can estimate if this is true is to check the final
     * position.  If not at the end...
     *****************************************************/
    if (pVidInfo->lImgBytePos !=
                            (LONG)pVidInfo->ulImgTotalBytes)
        {
        lRetCode = MMIO_WARNING;
        }
    /*****************************************************
     * Set up width and height of image in the buffer.
     *****************************************************/
    ulHeight = pVidInfo->mmImgHdr.mmXDIBHeader.BMPInfoHeader2.cy;
    usBitCount =
        pVidInfo->mmImgHdr.mmXDIBHeader.BMPInfoHeader2.cBitCount;

    /*****************************************************
     * Get the line width in YUV pels and packed bytes.
     *****************************************************/
    ulYUVPelWidth = pVidInfo->mmotHeader.mmXlen;
    lYUVBytesPerLine = (LONG)(ulYUVPelWidth * 3) >> 1;

    /*****************************************************
     * This changes from OS/2 PM bottom-up bitmap form
     *    to M-Motion's top-down form.  Flip all pad, boundary
     *    bytes as well
     *****************************************************/
    ImgBufferFlip ((PBYTE)pVidInfo->lpImgBuf,
                           pVidInfo->ulImgPaddedBytesPerLine,
                           ulHeight);

    /*****************************************************
     * Determine number of POSSIBLE pels on a line, though
     *   some may be overflow in 1bpp and 4bpp modes.
     *
     * From that, we can calc the number of RGB pels we will
```

```
 *   create to represent this line.
 ****************************************************/
ulImgPelWidth = (pVidInfo->ulImgPelBytesPerLine << 3) /
                            usBitCount;
lRGBBytesPerLine = ulImgPelWidth * 3;

/****************************************************
 * Ensure the width is on a 4-pel boundary, necessary for
 *     M-Motion.  We will buffer with black
 *    *** THIS IS ONLY NECESSARY FOR M-MOTION IMAGES
 ****************************************************/
if (ulImgPelWidth % 4)
    ulMaxPelWidth = (((ulImgPelWidth >> 2) + 1) << 2);
else
    ulMaxPelWidth = ulImgPelWidth;

/* #RGB bytes/line = #pels * 3 bytes/pel */
ulRGBMaxBytesPerLine = ulMaxPelWidth * 3;


/****************************************************
 * Create a buffer for one line of RGB data, accounting for
 *   the 4-pel boundary required.  Extra bytes won't be used.
 ****************************************************/
if (DosAllocMem ((PPVOID) &lpRGBLine,
                 ulRGBMaxBytesPerLine,
                 fALLOC))
    return (MMIO_ERROR);

/****************************************************
 * Create a buffer for one line of YUV data.
 ****************************************************/
if (DosAllocMem ((PPVOID) &lpYUVLine,
                 lYUVBytesPerLine,
                 fALLOC))
    return (MMIO_ERROR);

/****************************************************
 * Zero out RGB buffer to cover for any extra black pels
 * needed at the end.
 ****************************************************/
memset (lpRGBLine, 0, ulRGBMaxBytesPerLine);

/****************************************************
 * Initialize start position of RGB buffer.
 ****************************************************/
lpImgBufPtr = pVidInfo->lpImgBuf;

/****************************************************
 * Process Image Buffer - Save to file
 * Place "your" processing code here, if full image
 *      buffering is performed.
 * For M-Motion:
 *    Loop
 *       1.Convert and copy a line of 1bpp, 4bpp, 8 bpp or
 *            24bpp data into a temporary 24 bpp RGB line.
 *            This line may contain overflow garbage
 *            pels from 1bpp and 4bpp modes (where
 *            width does not fall on even byte boundaries.)
 *       2.Convert the temporary RGB line contents into a
 *           YUV line.  ONLY that data necessary converted.
 *           Overflow from bitmap data ignored.
 *       3.Write the YUV line to the file
 ****************************************************/
for (ulRowCount = 0;
     ulRowCount < ulHeight;
     ulRowCount++)
    {
    /************************************************
     * Convert 1 line of Image data into RGB data
     ************************************************/
   switch (usBitCount)
       {
       case 1:
           {
           /* Convert 1bpp padded image buffer into 24-bit */
           /*    RGB line buffer, w/pads                   */
           Convert1BitTo24Bit (
                   (PBYTE)lpImgBufPtr,
```

```
                (PRGB) lpRGBLine,
                (PRGB) &(pVidInfo->rgbPalette),
                pVidInfo->ulImgPelBytesPerLine);
            break;
            }

        case 4:
            {
            /* Convert data from app buffer into 24-bit and */
            /*    copy into image buffer                    */
            Convert4BitTo24Bit (
                    (PBYTE)lpImgBufPtr,
                    (PRGB) lpRGBLine,
                    (PRGB) &(pVidInfo->rgbPalette),
                    pVidInfo->ulImgPelBytesPerLine);
            break;
            }

        case 8:
            {
            /* Convert data from app buffer into 24-bit and */
            /*    copy into image buffer                    */
            Convert8BitTo24Bit (
                    (PBYTE)lpImgBufPtr,
                    (PRGB) lpRGBLine,
                    (PRGB) &(pVidInfo->rgbPalette),
                    pVidInfo->ulImgPelBytesPerLine);
            break;
            }

        case 24:
            {
            /* Copy raw RGB data from the image buffer into */
            /*    the temporary                             */
            /*   RGB line.  Only copy those pels necessary. */
            /*    No conversion required                    */
            memcpy ((PVOID) lpRGBLine,
                           (PVOID) lpImgBufPtr,
                           ulYUVPelWidth * 3);
            break;
            }

        }   /* end of Switch for Bit Conversion block */

     /************************************************
      * Convert one line at a time from RGB to YUV.
      ************************************************/
     ConvertOneLineRGBtoYUV (lpRGBLine,
                             lpYUVLine,
                             ulYUVPelWidth);

     /************************************************
      * Write out line of YUV data to the file.
      ************************************************/
     lBytesWritten  = mmioWrite (pVidInfo->hmmioSS,
                                 (PVOID) lpYUVLine,
                                 lYUVBytesPerLine);

     /* Check if error or EOF  */
     if (lBytesWritten != lYUVBytesPerLine)
        {
        lRetCode = lBytesWritten;
        break;
        }

     /************************************************
      * Make sure bitmap image buffer pointer is correct
      *    for next line to be converted.  Move forward ALL
      *    the bytes in the bitmap line, including overflow
      *    and pad bytes.
      ************************************************/
     lpImgBufPtr += pVidInfo->ulImgPaddedBytesPerLine;
     }

/*****************************************************
 * Free temp buffers.
 *****************************************************/
if (lpRGBLine)
   {
```

```
           DosFreeMem ((PVOID) lpRGBLine);
           }

        if (lpYUVLine)
           {
           DosFreeMem ((PVOID) lpYUVLine);
           }
        }  /* end IF WRITE & IMAGE BUFFER block */

    /*********************************************************
     * Free the RGB buffer, if it exists, that was created
     * for the translated READ operations.
     *********************************************************/
    if (pVidInfo->lpRGBBuf)
       {
       DosFreeMem ((PVOID) pVidInfo->lpRGBBuf);
       }

    /*********************************************************
     * Free the IMG buffer, if it exists, that was created
     * for the translated  WRITE operations.
     *********************************************************/
    if (pVidInfo->lpImgBuf)
       {
       DosFreeMem ((PVOID) pVidInfo->lpImgBuf);
       }

    /*********************************************************
     * Close the file with mmioClose.
     *********************************************************/
    rc = mmioClose (pVidInfo->hmmioSS, 0);

    DosFreeMem ((PVOID) pVidInfo);

    if (rc != MMIO_SUCCESS)
       return (rc);

    return (lRetCode);
    }  /* end case of MMIOM_CLOSE */
```

MMIOM_IDENTIFYFILE

The IOProc determines how to handle this message. All IOProcs must support this message because mmioOpen sends MMIOM_IDENTIFYFILE when it attempts to automatically identify a file.

For file format IOProcs, the header needs to be read and checked to see if it matches with what the IOProc expects. The *lParm2* field contains the handle used for the mmioRead function. In the following example, the header is a M-Motion file that is compared with the expected string defined in the IOProc. If it compares correctly, the message returns TRUE; otherwise FALSE is returned.

```
case MMIOM_IDENTIFYFILE:
    {
    /*********************************************************
     * Declare local variables.
     *********************************************************/
    MMOTIONHEADER   mmotHeader;   /* M-Motion structure variable */
    HMMIO           hmmioTemp;    /* MMIO File Handle            */
    ULONG           ulWidth;
    ULONG           ulHeight;
    ULONG           ulRequiredFileLength;
    ULONG           ulActualFileLength;
    BOOL             fValidMMotionFile = FALSE;

    ULONG           ulTempFlags = MMIO_READ | MMIO_DENYWRITE |
MMIO_NOIDENTIFY;
                                  /* Flags used for temp open  */
                                  /* and close                 */

    /*********************************************************
     * We need either a file name (lParam1) or file handle (lParam2)
```

```
 ************************************************************/
if (!lParam1 && !lParam2)
    return (MMIO_ERROR);

/* Copy the file handle, assuming one was provided... */
hmmioTemp = (HMMIO)lParam2;

/************************************************************
 * If no handle, then open the file using the string name
 ************************************************************/
if (!hmmioTemp)
   {
   if (!(hmmioTemp = mmioOpen ((PSZ) lParam1,
                               NULL,
                               ulTempFlags)))
       {
       return (MMIO_ERROR);
       }
   }

/************************************************************
 * Read in enough bytes to check out file.
 ************************************************************/
if (sizeof (MMOTIONHEADER) !=
               mmioRead (hmmioTemp,
                         (PVOID) &mmotHeader,
                         (ULONG) sizeof (MMOTIONHEADER)))
    {
    /*******************************************************
     * Fail so close file and then return.
     *******************************************************/
    if (!lParam2) /* Don't close handle if provided to us  */
       mmioClose (hmmioTemp, 0);
    return (MMIO_ERROR);
    }

/************************************************************
 * Close file before returning.
 ************************************************************/
if (!lParam2) /* Don't close handle if provided to us  */
   mmioClose (hmmioTemp, 0);

/************************************************************
 * Check validity of file and return result.
 ************************************************************/
if (memcmp (mmotHeader.mmID, "YUV12C", 6) == 0)
    {
    ulWidth = mmotHeader.mmXlen;
    ulHeight = mmotHeader.mmYlen;

    /* Calculate what the length of the file SHOULD be based on the */
    /*   header contents                                             */
    ulRequiredFileLength = (((ulWidth >> 2) * 6) * ulHeight) +
                                          sizeof (MMOTIONHEADER);

    /* Query what the ACTUAL length of the file is                   */
    ulActualFileLength = (ULONG)mmioSeek (hmmioTemp, 0, SEEK_END);

    /* If these don't match, then it isn't a VALID M-Motion file     */
    /*     - regardless of what the header says.                     */
    if (ulRequiredFileLength == ulActualFileLength)
       fValidMMotionFile = TRUE;
    else
       fValidMMotionFile = FALSE;
    }  /* end header check block */

/************************************************************
 * Close file before returning.
 ************************************************************/
if (!lParam2)  /* Don't close handle if provided to us      */
   mmioClose (hmmioTemp, 0);

if (fValidMMotionFile)
   return (MMIO_SUCCESS);
else
   return (MMIO_ERROR);
} /* end case of MMIOM_IDENTIFYFILE */
```

MMIOM_GETFORMATINFO

This message requests information about the format of the IOProc. MMIO provides a list of MMFORMATINFO structures containing descriptive information about the formats supported by currently installed IOProcs; for example, the format name, the FOURCC identifier, and related information. If this message is not defined in the IOProc or the IOProc does not handle the request successfully, mmioGetFormatInfo creates a blank MMFORMATINFO structure and attaches it to the internal list. It is recommended to hard-code the actual format information in the IOProc message handler code for the *ulStructLen* , *fccIOProc* , *ulMediaType* , and *ulFlags* fields. In addition, store other information (*ulCodePage* , *ulLanguage* , *lNameLength* , and *aulDefaultFormatExt* if any) in a resource file for NLS considerations.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_GETFORMATINFO message.

```
case MMIOM_GETFORMATINFO:
    {
    /*********************************************************
     * Declare local variables.
     *********************************************************/
    PMMFORMATINFO       pmmformatinfo;

    /*********************************************************
     * Set pointer to MMFORMATINFO structure.
     *********************************************************/
    pmmformatinfo = (PMMFORMATINFO) lParam1;

    /*********************************************************
     * Fill in the values for the MMFORMATINFO structure.
     *********************************************************/
    pmmformatinfo->ulStructLen  = sizeof (MMFORMATINFO);
    pmmformatinfo->fccIOProc    = FOURCC_MMOT;
    pmmformatinfo->ulIOProcType = MMIO_IOPROC_FILEFORMAT;
    pmmformatinfo->ulMediaType  = MMIO_MEDIATYPE_IMAGE;

    pmmformatinfo->ulFlags      = MMIO_CANREADTRANSLATED        |
                                  MMIO_CANREADUNTRANSLATED      |
                                  MMIO_CANWRITETRANSLATED       |
                                  MMIO_CANWRITEUNTRANSLATED     |
                                  MMIO_CANREADWRITEUNTRANSLATED |
                                  MMIO_CANSEEKTRANSLATED        |
                                  MMIO_CANSEEKUNTRANSLATED;

    strcpy ((PSZ) pmmformatinfo->szDefaultFormatExt, pszMotionExt);
    if (GetNLSData( &pmmformatinfo->ulCodePage,
                    &pmmformatinfo->ulLanguage ))
       {
       return( -1L );
       }

    if (GetFormatStringLength( FOURCC_MMOT,
                               &(pmmformatinfo->lNameLength) ))
       {
       return( -1L );
       }


    /*********************************************************
     * Return success back to the application.
     *********************************************************/
    return (MMIO_SUCCESS);
    } /* end case of MMIOM_GETFORMATINFO */
```

MMIOM_GETFORMATNAME

This message requests the descriptive format name supported by the IOProc. It is recommended to contain the strings in resource files for

NLS considerations.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_GETFORMATNAME message.

```
case MMIOM_GETFORMATNAME:
    {
    LONG lBytesCopied;

    /*********************************************************
     * Copy the M-Motion format string into buffer supplied by
     * lParam1.  Only put in the amount of my string up to the
     * allocated amount which is in lParam2.  Leave enough room
     * for the NULL termination.
     *********************************************************/
    lBytesCopied = GetFormatString( FOURCC_MMOT,
                                    (char *)lParam1,
                                    lParam2 );

    return (lBytesCopied);
    } /* end case of MMIOM_GETFORMATNAME */
```

MMIOM_QUERYHEADERLENGTH

This message requests the IOProc to return the size of the header for the current file or file element opened by mmioOpen. The mmioQueryHeaderLength function issues a MMIOM_QUERYHEADERLENGTH message to determine the buffer size that is needed by mmioGetHeader to obtain header data. This is necessary because headers vary in length.

To implement this message, save the current file position by using mmioSeek, then issue a call to mmioRead to read the size of the header into a buffer. The read can be done without a seek because mmioQueryHeaderLength saves the current file position when the call is issued. It seeks the file to its beginning, then seeks it back to the saved file position after the IOProc is called. The use of mmioRead is important here if the file is using buffered I/O so that all of MMIO's internal data fields are properly maintained throughout the message processing. It will allow subsequent file reads after this message is called to occur at the proper place in the file.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_QUERYHEADERLENGTH message.

```
case MMIOM_QUERYHEADERLENGTH:
    {
    /*********************************************************
     * If there is no MMIOINFO block then return an error.
     *********************************************************/
    if (!pmmioinfo)
        return (0);

    /*********************************************************
     * If header is in translated mode then return the media
     * type specific structure size.
     *********************************************************/
    if (pmmioinfo->ulTranslate & MMIO_TRANSLATEHEADER)
        return (sizeof (MMIMAGEHEADER));

    else
        /*****************************************************
         * Header is not in translated mode so return the size
         * of the M-Motion header.
         *****************************************************/
        return (sizeof (MMOTIONHEADER));

    break;
    } /* end case of MMIOM_QUERYHEADERLENGTH */
```

MMIOM_GETHEADER

This message requests the IOProc to return header-specific information about the current file or file element opened for reading by mmioOpen; for example, media type, media structure, and the size of the media structure. When you call mmioRead to read in the file header, use the size of the header, which is passed in by the *lParam2* parameter.

When the translate header is TRUE, the IOProc is expected to return the standard presentation header structure for that media type. The IOProc must transpose the file's native header data into that structure.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_GETHEADER message.

```
case MMIOM_GETHEADER:
    {
    /*********************************************************
     * Declare local variables.
     *********************************************************/
    PMMFILESTATUS       pVidInfo;

    /*********************************************************
     * Check for valid MMIOINFO block.
     *********************************************************/
    if (!pmmioinfo)
        return (0);

    /*********************************************************
     * Set up our working file status variable.
     *********************************************************/
    pVidInfo = (PMMFILESTATUS)pmmioinfo->pExtraInfoStruct;

    /************************************************
     * Getheader only valid in READ or READ/WRITE mode.
     * There is no header to get in WRITE mode.  We
     * must also have a valid file handle to read from
     ************************************************/
    if ((pmmioinfo->ulFlags & MMIO_WRITE) ||
        (!(pVidInfo->hmmioSS)))
       return (0);

    /*********************************************************
     * Check for Translation mode.
     *********************************************************/
    if (!(pmmioinfo->ulTranslate & MMIO_TRANSLATEHEADER))
        {
        /*****************************************************
         * Translation is off.
         *****************************************************/
        if (lParam2 < sizeof (MMOTIONHEADER))
            {
            pmmioinfo->ulErrorRet = MMIOERR_INVALID_BUFFER_LENGTH;
            return (0);
            }

        if (!lParam1)
            {
            pmmioinfo->ulErrorRet = MMIOERR_INVALID_STRUCTURE;
            return (0);
            }

        /*****************************************************
         * Read in first 16 bytes to fill up M-Motion header.
         *****************************************************/
        memcpy ((PVOID) lParam1,
                (PVOID)&pVidInfo->mmotHeader,
                sizeof (MMOTIONHEADER));

        /* Indicate that the header has been set, which    */
        /* is meaningless in read mode, but allows the     */
        /* application to do writes in read/write mode     */
        pVidInfo->bSetHeader = TRUE;

        return (sizeof (MMOTIONHEADER));
        }   /* end IF NOT TRANSLATED block */

    /*****************
     * TRANSLATION IS ON
     *****************/
    if (lParam2 < sizeof (MMIMAGEHEADER))
```

```
        {
        pmmioinfo->ulErrorRet = MMIOERR_INVALID_BUFFER_LENGTH;
        return (0);
        }

    if (!lParam1)
        {
        pmmioinfo->ulErrorRet = MMIOERR_INVALID_STRUCTURE;
        return (0);
        }

    memcpy ((PVOID)lParam1,
            (PVOID)&pVidInfo->mmImgHdr,
            sizeof (MMIMAGEHEADER));

    return (sizeof (MMIMAGEHEADER));
    } /* end case of MMIOM_GETHEADER */
```

MMIOM_SETHEADER

This message requests the IOProc to set header-specific information in a file opened for writing by mmioOpen. This would include data such as resolution and colors for images, and duration and sample rate for audio.

When the translate header is TRUE, the IOProc should expect to be passed the standard presentation header for that media type. The IOProc is expected to transpose the data from that structure into its native header structure before writing the header to the file.

The following example shows an example of how the M-Motion IOProc supports the MMIOM_SETHEADER message.

```
case MMIOM_SETHEADER:
     {
     /*********************************************************
      * Declare local variables.
      *********************************************************/
     PMMIMAGEHEADER          pMMImgHdr;
     PMMFILESTATUS           pVidInfo;
     USHORT                  usNumColors;
     ULONG                   ulImgBitsPerLine;
     ULONG                   ulImgBytesPerLine;
     ULONG                   ulBytesWritten;
     ULONG                   ulWidth;
     ULONG                   ul4PelWidth;
     ULONG                   ulHeight;
     USHORT                  usPlanes;
     USHORT                  usBitCount;
     USHORT                  usPadBytes;

     /*********************************************************
      * Check for valid MMIOINFO block.
      *********************************************************/
     if (!pmmioinfo)
         return (MMIO_ERROR);

     /*********************************************************
      * Set up our working variable MMFILESTATUS.
      *********************************************************/
     pVidInfo = (PMMFILESTATUS) pmmioinfo->pExtraInfoStruct;

     /*********************************************************
      * Only allow this function if we are in WRITE mode
      * And only if we have not already set the header
      *********************************************************/
     if ((!(pmmioinfo->ulFlags & MMIO_WRITE)) ||
         (!(pVidInfo->hmmioSS)) ||
         (pVidInfo->bSetHeader))
         return (0);

     /*********************************************************
      * Make sure lParam1 is a valid pointer
      *********************************************************/
```

```
if (!lParam1)
    {
    pmmioinfo->ulErrorRet = MMIOERR_INVALID_STRUCTURE;
    return (0);
    }

/*********************************************************
 * Header is not in translated mode.
 *********************************************************/
if (!(pmmioinfo->ulTranslate & MMIO_TRANSLATEHEADER))
    {
    /*****************************************************
     * Make sure lParam2 is correct size
     *****************************************************/
    if (lParam2 != MMOTION_HEADER_SIZE)
        {
        pmmioinfo->ulErrorRet = MMIOERR_INVALID_BUFFER_LENGTH;
        return (0);
        }

    /*****************************************************
     * Ensure that the header at least begins with "YUV12C"
     *****************************************************/
    if (strncmp ((char *)lParam1, "YUV12C", 6))
       {
       pmmioinfo->ulErrorRet = MMIOERR_INVALID_STRUCTURE;
       return (0);
       }

    /*****************************************************
     * Take 16 byte buffer (lParam1), write to file and .
     *    copy to internal structure.
     *****************************************************/
    memcpy ((PVOID)&pVidInfo->mmotHeader,
            (PVOID)lParam1, (ULONG) MMOTION_HEADER_SIZE);
    ulBytesWritten = mmioWrite (pVidInfo->hmmioSS,
                                (PVOID) lParam1,
                                (ULONG) MMOTION_HEADER_SIZE);

    /*********************************
     * Check for an error on the write..
     *********************************/
    if (ulBytesWritten != MMOTION_HEADER_SIZE)
        return (0);   /* 0 indicates error */

    /*********************************
     * Success...
     *********************************/
    pVidInfo->bSetHeader = TRUE;
    return (sizeof (MMOTIONHEADER));
    }  /* end IF NOT TRANSLATED block */

/*********************************************************
 * Header is translated.
 *********************************************************/

/*********************************************************
 * Create local pointer media specific structure.
 *********************************************************/
pMMImgHdr = (PMMIMAGEHEADER) lParam1;


/*********************************************************
 * Check for validity of header contents supplied
 *********************************************************
 *  --  Length must be that of the standard header
 *  --  NO Compression
 *      1 plane
 *      24, 8, 4 or 1 bpp
 *********************************************************/
usBitCount = pMMImgHdr->mmXDIBHeader.BMPInfoHeader2.cBitCount;
if ((pMMImgHdr->mmXDIBHeader.BMPInfoHeader2.ulCompression !=
     BCA_UNCOMP) ||
    (pMMImgHdr->mmXDIBHeader.BMPInfoHeader2.cPlanes != 1) ||
    (! ((usBitCount == 24) || (usBitCount == 8) ||
        (usBitCount == 4) || (usBitCount == 1)))
   )
    {
    pmmioinfo->ulErrorRet = MMIOERR_INVALID_STRUCTURE;
```

```
    return (0);
    }

if (lParam2 != sizeof (MMIMAGEHEADER))
   {
   pmmioinfo->ulErrorRet = MMIOERR_INVALID_BUFFER_LENGTH;
   return (0);
   }

/*********************************************************
 * Complete MMOTIONHEADER.
 *********************************************************/
memcpy ((PVOID)&pVidInfo->mmotHeader.mmID, "YUV12C", 6);
pVidInfo->mmotHeader.mmXorg  = 0;
pVidInfo->mmotHeader.mmYorg  = 0;

/*******************************************************
 * Ensure we will save stuff on 4-pel boundaries when
 * we actually convert to YUV and pack the bits.
 * We don't change what the user is actually going to
 * give us.  The user thinks he is on 1-pel boundaries,
 * and that is how we buffer the RGB data.
 *******************************************************/
ulWidth  = pMMImgHdr->mmXDIBHeader.BMPInfoHeader2.cx;
ulHeight = pMMImgHdr->mmXDIBHeader.BMPInfoHeader2.cy;
if (ulWidth % 4)
    ul4PelWidth = (((ulWidth >> 2) + 1) << 2);
else
    ul4PelWidth = ulWidth;
pVidInfo->mmotHeader.mmXlen  = (USHORT) ul4PelWidth;
pVidInfo->mmotHeader.mmYlen  = (USHORT) ulHeight;

/*********************************************************
 * Write the M-Motion Header.
 *********************************************************/
ulBytesWritten = mmioWrite (pVidInfo->hmmioSS,
                            (PVOID) &pVidInfo->mmotHeader,
                            (ULONG) MMOTION_HEADER_SIZE);

/*************************************
 * Check for an error on the write...
 *************************************/
if (ulBytesWritten != MMOTION_HEADER_SIZE)
    return (0);

/*********************************************************
 * Flag that MMIOM_SETHEADER has been done.  It can only
 * be done ONCE for a file.  All future attempts will
 * be flagged as errors.
 *********************************************************/
pVidInfo->bSetHeader = TRUE;

/*********************************************************
 * Create copy of MMIMAGEHEADER for future use.
 *********************************************************/
pVidInfo->mmImgHdr = *pMMImgHdr;

/*********************************************************
 * Check bitcount, set palette if less than 24.
 *********************************************************/
if (usBitCount < 24)
   {
   /*****************************************************
    * Find out how many colors are in the palette.
    *****************************************************/
   usNumColors = (USHORT)(1 << usBitCount);

   /*****************************************************
    * Take the RGB2 palette and convert it to an RGB palette
    *    Place the converted palette in MMFILESTATUS struct
    *****************************************************/
   RGB2_To_RGB (pVidInfo->mmImgHdr.bmiColors,
                (PRGB) &(pVidInfo->rgbPalette),
                usNumColors);
   }

/*******************************************************
 * We must allocate the buffer.  The app will load the
 * buffer on subsequent write calls.
```

```
 ******************************************************/
usPlanes = pVidInfo->mmImgHdr.mmXDIBHeader.BMPInfoHeader2.cPlanes;

/*****************************************************
 * Determine total Image size
 *****************************************************
 * Find bits-per-line BEFORE padding and 1bpp or 4bpp pel overflow
 *****************************************************/
ulImgBitsPerLine = ulWidth * usPlanes * usBitCount;
ulImgBytesPerLine = ulImgBitsPerLine >> 3;

/*****************************************************
 * Account for extra pels not on an even byte boundary
 *   for 1bpp and 4bpp
 *****************************************************/
if (ulImgBitsPerLine % 8)
   ulImgBytesPerLine ++;

pVidInfo->ulImgPelBytesPerLine = ulImgBytesPerLine;

/*****************************************************
 * Ensure the row length in bytes accounts for byte padding.
 *   All bitmap data rows are aligned on LONG/4-BYTE boundaries.
 *   The data FROM an application should always appear in this form
 *****************************************************/
usPadBytes = (USHORT)(ulImgBytesPerLine % 4);
if (usPadBytes)
   ulImgBytesPerLine += 4 - usPadBytes;

pVidInfo->ulImgPaddedBytesPerLine  = ulImgBytesPerLine;
pVidInfo->ulImgTotalBytes = ulImgBytesPerLine * ulHeight;

/*****************************************************
 * Get space for full image buffer.
 *****************************************************/
if (DosAllocMem ((PPVOID) &(pVidInfo->lpImgBuf),
                 pVidInfo->ulImgTotalBytes,
                 fALLOC))
    return (MMIO_ERROR);

/*****************************************************
 * Set up initial pointer value within RGB buffer & image
 *****************************************************/
pVidInfo->lImgBytePos = 0;

return (sizeof (MMIMAGEHEADER));
}  /* end case of MMIOM_SETHEADER */
```

CODEC Support

The following sections examine the Ultimotion IOProc (ULIOT) sample (located in the \TOOLKIT\SAMPLES\MM\ULTIMOIO subdirectory), which includes source code to compress or decompress a data object. The ULIOT I/O procedure calls the Ultimotion CODEC procedure to compress raw digital images into a smaller form so they can use less storage space. To play motion video, the ULIOT sample can also call the decompressor part of the CODEC procedure to reconstruct the original image from the compressed data.

Decompression

The following section illustrates how to decompress an image to play motion video. Follow these steps for decompression support:

1. Open the file and determine which CODEC(s) are required.
2. Load the CODEC DLL file.
3. Load and initialize the CODEC procedure.
4. Call the CODEC to decompress data.
5. Close the CODEC and release resources after use.

Opening an Image Object

To play back or *decompress* an image object, you must first open the movie file to find the four-character code (FOURCC) of the compression type and initiate the track information (as shown in the following example). The open routine allows for multiple compression types per stream.

```
LONG IOProcOpen (PMMIOINFO pmmioinfo, PSZ pszFileName) {
   LONG            rc = MMIO_SUCCESS;        /* Return code.           */
   LONG            lFilePosition;            /* Logical file position. */
   MMIOINFO        Localmmioinfo;            /* For locally used.      */
   PINSTANCE       pinstance;                /* Local work structure.  */

   if (pmmioinfo == NULL) return MMIO_ERROR;

   if (CheckMem((PVOID)pmmioinfo, sizeof(MMIOINFO), PAG_WRITE))
      return MMIO_ERROR;

   /*****************************************************************/
   /* Validate the open flags for this File Format IOProc           */
   /* (INVALID_OPEN_FLAGS should be defined in the ff.h - file format  */
   /*  specific header file.)                                       */
   /*****************************************************************/

   if (pmmioinfo->ulFlags  & INVALID_OPEN_FLAGS) {
       pmmioinfo->ulErrorRet = MMIOERR_INVALID_ACCESS_FLAG;
       return(MMIO_ERROR);
   }

   ENTERCRITX;
   if ((pinstance = (PINSTANCE)HhpAllocMem(hheap,sizeof(INSTANCE)))
         == NULL) {
       EXITCRIT;
       pmmioinfo->ulErrorRet = MMIOERR_OUTOFMEMORY;
       return(MMIO_ERROR);                 /* Allocate work struct. */
   }
   EXITCRIT;

   pmmioinfo->pExtraInfoStruct = (PVOID) pinstance;
   pmmioinfo->fccIOProc = HEX_FOURCC_FFIO; /* Set for CODEC loading. */
   ioInstanceInit(pinstance);

   /* Validate read flags before doing read initialization */

   if (( pmmioinfo->ulFlags & MMIO_READ ) &&
               !( pmmioinfo->ulFlags & INVALID_READ_FLAGS )) {

      /* IOProc identifies Storage System */

      memcpy (&Localmmioinfo, pmmioinfo, sizeof(MMIOINFO));
      Localmmioinfo.pIOProc = NULL;
      Localmmioinfo.fccIOProc = pmmioinfo->fccChildIOProc;
      Localmmioinfo.ulFlags |= MMIO_NOIDENTIFY; /* Eliminate callbacks */
      Localmmioinfo.ulFlags &= ~MMIO_ALLOCBUF;  /* Force non-buffered open */

      rc = ioIdentifyStorageSystem(&Localmmioinfo,pszFileName);

      if (rc != MMIO_SUCCESS) {        /* if error,                 */
         ioCleanUp(pmmioinfo);
         return(rc);
      }

      /*****************************************************/
      /* Allocate memory for pTempBuffer which is used when */
      /* IOProcReadInterLeaved is called.                  */
      /*****************************************************/

      if (ENTERCRIT(rc)) {
         ioCleanUp(pmmioinfo);
         return(rc);
```

```
      }

      if ((pinstance->pTempBuffer = HhpAllocMem(hheap, DEFAULTBUFFERSIZE))
         == NULL) {
         EXITCRIT;
         ioCleanUp(pmmioinfo);
         return(MMIOERR_OUTOFMEMORY);
      }
      EXITCRIT;
      pinstance->ulTempBufferSize = DEFAULTBUFFERSIZE;

      /*************************************************/
      /* Open Movie file                               */
      /*************************************************/

      if ( pmmioinfo->fccChildIOProc != FOURCC_MEM ) {
         Localmmioinfo.cchBuffer = 0;
         Localmmioinfo.pchBuffer = NULL;
      }
      pinstance->hmmioFileHandle = mmioOpen(pszFileName,
         &Localmmioinfo,MMIO_NOIDENTIFY);
           /* Test file open error.*/
      if (pinstance->hmmioFileHandle <= (HMMIO)0L) {
         rc = Localmmioinfo.ulErrorRet;
      }

      /*************************************************/
      /* Call file format specific open routine        */
      /*************************************************/

      else if (!(rc = ffOpenRead(pmmioinfo, pinstance))) {
         if(!(rc = ioAddTracksToMovieHeader(pinstance))) {

            /*********************************************************/
            /* Set lLogicalFilePos to a position pass the header     */
            /* block to allow read occurring at the first byte of    */
            /* non-header data.                                      */
            /*********************************************************/
            lFilePosition = ffSeekToDataBegin(pmmioinfo,pinstance);
            if (lFilePosition < MMIO_SUCCESS)
               rc = MMIO_ERROR;
            else
               pinstance->lFileCurrentPosition = lFilePosition;
         }
      }

      if (rc) {
         ioCleanUp(pmmioinfo);
         return(rc);
      }
   }

   /* Validate Write flags before doing initialization */

#ifndef WORKSHOP

   if ((pmmioinfo->ulFlags & (MMIO_READWRITE | MMIO_WRITE)) &&
               !(pmmioinfo->ulFlags & INVALID_WRITE_FLAGS)) {

      /* Open the movie file */

      memset (&Localmmioinfo, '\0', sizeof(MMIOINFO));
      Localmmioinfo.pIOProc   = NULL;
      Localmmioinfo.fccIOProc = pmmioinfo->fccChildIOProc;

      if (pmmioinfo->fccChildIOProc != FOURCC_MEM) {
         Localmmioinfo.cchBuffer = 0;
         Localmmioinfo.pchBuffer = NULL;
      }

      Localmmioinfo.ulFlags |= MMIO_NOIDENTIFY; /* Eliminate callbacks */
      Localmmioinfo.ulFlags &= ~MMIO_ALLOCBUF;  /* Force non-buffered open. */
                                                /* MMIO may do buffering. */

      pinstance->hmmioFileHandle = mmioOpen(pszFileName, &Localmmioinfo,
                                       MMIO_READWRITE | MMIO_NOIDENTIFY);

      if (pinstance->hmmioFileHandle <= (HMMIO)0L) /* Test file open error. */
         rc = Localmmioinfo.ulErrorRet;
```

```
      else
         /* Call file format specific open routine */
         rc = ffOpenWrite(pmmioinfo, pinstance);

      if (rc != 0) {
         ioCleanUp(pmmioinfo);
         return(rc);
      }
   }

#else    /* WORKSHOP next */

   if ((pmmioinfo->ulFlags & (MMIO_READWRITE | MMIO_WRITE)) &&
             !(pmmioinfo->ulFlags & INVALID_WRITE_FLAGS)) {

      /* Open the movie file */

      memset (&Localmmioinfo, '\0', sizeof(MMIOINFO));
      Localmmioinfo.pIOProc   = NULL;
      Localmmioinfo.fccIOProc = pmmioinfo->fccChildIOProc;
      Localmmioinfo.ulFlags  = pmmioinfo->ulFlags;
      Localmmioinfo.ulFlags |= MMIO_NOIDENTIFY; /* Eliminate callbacks */
      Localmmioinfo.ulFlags &= ~MMIO_ALLOCBUF;  /* Force non-buffered open. */
                                                /* MMIO may do buffering. */

      if (!(pmmioinfo->ulFlags & MMIO_CREATE)) {
         rc = ioIdentifyStorageSystem(&Localmmioinfo, pszFileName);

         if (rc != MMIO_SUCCESS) {        /* if error */
            pmmioinfo->ulErrorRet = rc;   /* see IdentifyStorageSystem */
            ioCleanUp(pmmioinfo);
            return(MMIO_ERROR);
         }

         /* Allocate memory for pTempBuffer which is used when */
         /* IOProcReadInterLeaved is called. */

         if (ENTERCRIT(rc)) {
            ioCleanUp(pmmioinfo);
            return MMIO_ERROR;
         }

         pinstance->pTempBuffer = HhpAllocMem(hheap, DEFAULTBUFFERSIZE);
         if (pinstance->pTempBuffer == NULL) {
            EXITCRIT;
            pmmioinfo->ulErrorRet = MMIOERR_OUTOFMEMORY;
            ioCleanUp(pmmioinfo);
            return MMIO_ERROR;
         }
         EXITCRIT;

         pinstance->ulTempBufferSize = DEFAULTBUFFERSIZE;
      }

      pinstance->lFileCurrentPosition = 0;

      pinstance->hmmioFileHandle = mmioOpen(pszFileName, &Localmmioinfo,
          Localmmioinfo.ulFlags);

      if (pinstance->hmmioFileHandle <= (HMMIO)0L) /* Test file open error. */
         rc = Localmmioinfo.ulErrorRet;
      else {
             /* Call file format specific open routine */
         rc = ffOpenWrite(pmmioinfo, pinstance);

         if (rc == 0) {
            if (!(pmmioinfo->ulFlags & MMIO_CREATE)) {
               rc = ioAddTracksToMovieHeader(pinstance);

               if (rc == 0) {

                  /* Set lLogicalFilePos to a position pass the header */
                  /* block to allow read occurring at the first byte */
                  /* of non-header data. */

                  lFilePosition = ffSeekToDataBegin(pmmioinfo, pinstance);
                  if (lFilePosition < MMIO_SUCCESS) rc = MMIO_ERROR;
                  else pinstance->lFileCurrentPosition = lFilePosition;
               }
```

```
            }
         }
      }

      if (rc != 0) {
         pmmioinfo->ulErrorRet = rc;
         ioCleanUp(pmmioinfo);
         return MMIO_ERROR;
      }
   }

   /* Set up the pathname in the instance structure */

   if (strlen(pszFileName) < CCHMAXPATH) {
      strcpy((PSZ)&(pinstance->szFileName), pszFileName);
      if ((pinstance->szFileName)[1] == ':')
         pinstance->ulEditFlags |= FULLY_QUALIFIED_PATH;
   }

#endif

   return MMIO_SUCCESS;
}
```

Determining the CODEC Procedure

Once the FOURCC is obtained, the appropriate CODEC procedure is determined and loaded. The following example illustrates the ioDetermineCodec routine, which determines which CODEC procedure to load. This routine first queries the hardware to determine what CODEC mode the hardware requires. Next, it queries the MMPMMMIO.INI file to determine if there is a CODEC procedure that matches the FOURCC, compression type, and capability flags received from the open routine. If no default CODEC is found, this routine queries the MMPMMMIO.INI file to find a compression type that works for this particular hardware. If there is no CODEC procedure known to the INI file that will match the FOURCC, the code searches internal CODEC tables for the necessary CODEC to load. The ULIOT sample provides an internal CODEC table located in the ULGDAT.C file (shown in the following example). This table is hard coded in the IOProc to specify what CODEC procedure to call.

```
LONG ioDetermineCodec ( PINSTANCE pinstance,
                        ULONG ulSearchFlags,
                        PCODECINIFILEINFO pcifi )

{
   LONG              rc = MMIO_SUCCESS; /* Return code of IOProcs call */
   USHORT            i;                 /* Loop index                  */
   ULONG             ulFlags;
   HPS               hps;               /* Use to query color support  */
   HAB               hab;               /* anchor block                */
   HMQ               hmq;               /* anchor block                */


   if (pcifi->ulCapsFlags & CODEC_DECOMPRESS) {
      /* Query the display mode */
      if (ulNumColors == 0) {       /* Get this info once per process */
         hab  = WinInitialize(0);
//       hmq  = WinCreateMsgQueue( hab, 0L );

         hps  = WinGetPS(HWND_DESKTOP);
         DevQueryCaps ( GpiQueryDevice(hps),
                        CAPS_COLORS,
                        1L,
                        (PLONG)&ulNumColors);

         WinReleasePS (hps);
//       WinDestroyMsgQueue( hmq );
         WinTerminate (hab);
         }

      /* Set the color depth for the CODEC we want */
      if (ulNumColors == 16)
         pcifi->ulCapsFlags |= CODEC_4_BIT_COLOR;
      else if (ulNumColors > 256)
```

```
         pcifi->ulCapsFlags |= CODEC_16_BIT_COLOR;
      else  /* 256 and anything else */
         pcifi->ulCapsFlags |= CODEC_8_BIT_COLOR;
      }

   /*********************************************************************/
   /* Search for the DEFAULT CODEC of this type from the MMIO INI file */
   /*********************************************************************/
   pcifi->ulCapsFlags |= CODEC_DEFAULT;   /* Pick default */
   ulFlags = ulSearchFlags |
             MMIO_MATCHFOURCC |
             MMIO_MATCHCOMPRESSTYPE |
             MMIO_MATCHCAPSFLAGS |
             MMIO_MATCHFIRST |
             MMIO_FINDPROC;

   if (!(rc = mmioIniFileCODEC(pcifi,ulFlags))) {
      return(MMIO_SUCCESS);
      }

   /*********************************************************************/
   /* If no default, find first one and use it from the MMIO INI file  */
   /*********************************************************************/
   pcifi->ulCapsFlags &= ~CODEC_DEFAULT;
   ulFlags = ulSearchFlags |
             MMIO_MATCHFOURCC |
             MMIO_MATCHCOMPRESSTYPE |
             MMIO_MATCHCAPSFLAGS |
             MMIO_MATCHFIRST |
             MMIO_FINDPROC;

   /* Match the fourcc, compress type, caps flags */
   if (!(rc = mmioIniFileCODEC(pcifi,ulFlags))) {
      return(MMIO_SUCCESS);
      }

   /*********************************************************************/
   /* Search any internal CODEC tables for the necessary CODEC to load. */
   /* Note: This is used for debugging new CODECs that have not been    */
   /* added to the MMPMMMIO.INI file.                                   */
   /*********************************************************************/
   for (i = 0; i < NUMBER_CODEC_TABLE; i++) {

      if ((acifiTable[i].ulCompressType == pcifi->ulCompressType) &&
          ((acifiTable[i].ulCapsFlags & pcifi->ulCapsFlags)
              == pcifi->ulCapsFlags)) {

         *pcifi = acifiTable[i];         /* Copy contents */
         return(MMIO_SUCCESS);
         }
      }


   return(MMIOERR_CODEC_NOT_SUPPORTED);
}
```

The following example illustrates how to hardcode a CODEC INI file information structure in an IOProc if the table is not represented in the MMPMMMIO.INI file.

```
CODECINIFILEINFO acifiTable[] = {
   {
      sizeof(CODECINIFILEINFO),
      FOURCC_FFIO,
      "ULBDC4",
       /* szDCIODLLName[]-Decompression IOProc DLL name */
      "CodecEntry",
       /* szDCIOProcName[]-Decomp IOProc entry pt proc name */
      UM_VIDEO_COMPRESSION_TYPE_BH146,
       /* ulDecompressionType - ID of each decompression type */
      0L,
      MMIO_MEDIATYPE_DIGITALVIDEO,
      CODEC_DECOMPRESS+           /* ulCapsFlags - Capabilities Flag */
         CODEC_SELFHEAL+          /* ulCapsFlags - Capabilities Flag */
         CODEC_ORIGIN_UPPERLEFT+  /* ulCapsFlags - Capabilities Flag */
         CODEC_4_BIT_COLOR,       /* ulCapsFlags - Capabilities Flag */
      0,
```

```
      0,
      0,
      0,
      0,
      8,
      8,
      0,
   },
   {
      sizeof(CODECINIFILEINFO),
      FOURCC_FFIO,
      "ULBDC8",
        /* szDCIODLLName[]-Decompression IOProc DLL name */
      "CodecEntry",
        /* szDCIOProcName[]-Decomp IOProc entry pt proc name */
      UM_VIDEO_COMPRESSION_TYPE_BH146,
        /* ulDecompressionType - ID of each decompression type */
      0L,
      MMIO_MEDIATYPE_DIGITALVIDEO,
      CODEC_DECOMPRESS+              /* ulCapsFlags - Capabilities Flag */
         CODEC_SELFHEAL+             /* ulCapsFlags - Capabilities Flag */
         CODEC_ORIGIN_UPPERLEFT+     /* ulCapsFlags - Capabilities Flag */
         CODEC_MULAPERTURE+          /* ulCapsFlags - Capabilities Flag */
         CODEC_DIRECT_DISPLAY+       /* ulCapsFlags - Capabilities Flag */
         CODEC_8_BIT_COLOR,          /* ulCapsFlags - Capabilities Flag */
      0,
      0,
      0,
      0,
      0,
      8,
      8,
      0,
   },

   {
      sizeof(CODECINIFILEINFO),
      FOURCC_FFIO,
      "ULBDC16",
          /* szDCIODLLName[]-Decompression IOProc DLL name */
      "CodecEntry",
          /* szDCIOProcName[]-Decomp IOProc entry point proc name */
      UM_VIDEO_COMPRESSION_TYPE_BH146,
          /* ulDecompressionType - ID of each decompression type */
      0L,
      MMIO_MEDIATYPE_DIGITALVIDEO,
      CODEC_DECOMPRESS+              /* ulCapsFlags - Capabilities Flag */
         CODEC_SELFHEAL+             /* ulCapsFlags - Capabilities Flag */
         CODEC_ORIGIN_UPPERLEFT+     /* ulCapsFlags - Capabilities Flag */
         CODEC_DIRECT_DISPLAY+       /* ulCapsFlags - Capabilities Flag */
         CODEC_16_BIT_COLOR,         /* ulCapsFlags - Capabilities Flag */
      0,
      0,
      0,
      0,
      0,
      8,
      8,
      0,
   },
};
```

Loading the CODEC Procedure

Once a CODEC procedure is determined, the CODEC DLL is loaded and added to the linked list of CODECs currently loaded for this movie instance. The ioLoadCodecDLL routine shown in the following example first checks to see that the CODEC procedure is not already loaded in the list. If it is not loaded in the list, it sends a DOSLoadModule function to load the DLL and find the entry. Next, it initializes the CODEC procedure in the ffOpenCodec routine shown in the following example.

Note: Refer to the *OS/2 Presentation Manager Guide and Reference* for more information on the DosLoadModule function.

```
PCCB ioLoadCodecDLL ( PINSTANCE pinstance,
                      PCODECINIFILEINFO pcifi,
                      PULONG phCodec )

{
   LONG      rc = MMIO_SUCCESS;     /* Return code of IOProc's call. */
   SZ        szLoadError[260];           /* Error returns. */
   PCCB      pccbNew;
   PCCB      pccb;
   HMODULE   hmod = 0L;
   PMMIOPROC pmmioproc;
   ULONG     hCodec = 0L;


   /*********************************************************************/
   /* Search if the CCB entry has been created for the passed in       */
   /* pszDLLName and pszProcName,if yes, then sets pccb pointer of     */
   /* ptracki to that node.(different track may share a same CCB)      */
   /*********************************************************************/
   for (pccb = pinstance->pccbList; pccb; pccb = pccb->pccbNext) {

      if (!stricmp(pcifi->szDLLName,pccb->cifi.szDLLName)) {
         hCodec = pccb->hCodec;

         if (!stricmp(pcifi->szProcName,pccb->cifi.szProcName)) {
            /* Proc entry names match */
            return(pccb);
            }
         }
      } /* end loop */


   /*********************************************************************/
   /* The above searching cannot find the DCIO node; if a same        */
   /* DLLName was found, query IOProc address directly, else load     */
   /* module then query I/O address before allocates a new pccb node. */
   /*********************************************************************/
   if (DosLoadModule(szLoadError,
                     (ULONG)sizeof(szLoadError),
                     pcifi->szDLLName,
                     &hmod))   {
      /* Load Error - MMIOERR_INVALID_DLLNAME */
      return(NULL);
      }

   if (DosQueryProcAddr(hmod,
                        0L,
                        pcifi->szProcName,
                        (PFN *)&pmmioproc))   {
      /* Query Error - MMIOERR_INVALID_PROCEDURENAME */
      return(NULL);
      }

   /*********************************************************************/
   /* The above loading and querying was successful; then create a new */
   /* node for the DCIO.  If HhpAllocMem fails, call DosFreeModule to  */
   /* free DCIO module before returning error.                         */
   /*********************************************************************/
   if (ENTERCRIT(rc)) {
      return(NULL);
      }

   pccbNew = (PCCB)HhpAllocMem(hheap,(ULONG)sizeof(CCB));

   EXITCRIT;

   if(!pccbNew) {
      DosFreeModule(hmod);
      /* Allocate error - MMIOERR_OUTOFMEMORY */
      return(NULL);
      }

   /*********************************************************************/
```

```
   /* Assigns the Decompress IOProc information, which is in record map */
   /* table, to the new DCIO node.                                      */
   /*********************************************************************/
   pccbNew->cifi = *pcifi;
   pccbNew->hmodule = hmod;
   pccbNew->pmmioproc = pmmioproc;
   pccbNew->hCodec = 0L;
   pccbNew->ulLastSrcBuf = 0L;
   pccbNew->ulMisc1 = 0L;
   pccbNew->ulMisc2 = 0L;

   /*********************************************************************/
   /* Adds new node to the beginning of ccb list.                       */
   /*********************************************************************/
   pccbNew->pccbNext = pinstance->pccbList;
   pinstance->pccbList = pccbNew;

   *phCodec = hCodec;
   return(pccbNew);
}



/************************* START OF SPECIFICATIONS *****************/
/*                                                                 */
/* SUBROUTINE NAME:  ioLoadCodec                                   */
/*                                                                 */
/* DESCRIPTIVE NAME: Load a CODEC IOProc and add it the PCCB list  */
/*                                                                 */
/* FUNCTION: This function loads a CODEC IOProc and adds it to the */
/*  linked list of CODECs currently loaded for this movie instance. */
/*                                                                 */
/* NOTES:                                                          */
/*                                                                 */
/* ENTRY POINT: ioLoadCodec                                        */
/*   LINKAGE:   CALL FAR (00:32)                                   */
/*                                                                 */
/* INPUT:                                                          */
/*             PINSTANCE pinstance     - Instant structure         */
/*             PTRACKI ptracki         - Track specific information */
/*             PCODECINIFILEINFO pcifi - CODEC INI file information */
/*                                                                 */
/* EXIT-NORMAL:                                                    */
/*              pccb                                               */
/*                                                                 */
/* EXIT-ERROR:                                                     */
/*              0L                                                 */
/*                                                                 */
/* SIDE EFFECTS:                                                   */
/*                                                                 */
/************************* END OF SPECIFICATIONS *****************/
PCCB ioLoadCodec ( PINSTANCE pinstance,
                   PTRACKI ptracki,
                   PCODECINIFILEINFO pcifi )

{
   LONG      rc = MMIO_SUCCESS; /* Return code of IOProc's call.*/
   PCCB      pccbNew;
   ULONG     hCodec = 0L;



   if (pccbNew = ioLoadCodecDLL(pinstance,pcifi,&hCodec)) {

      /*****************************************************************/
      /* Open the Codec IOProc and save the hCodec in the pccb structure */
      /*****************************************************************/
      if (rc = ffOpenCodec(pinstance, pccbNew, hCodec, ptracki)) {

         pinstance->pccbList = pccbNew->pccbNext;   /* unlink from list */
         ioCloseCodec(pccbNew);
         pccbNew = NULL;  /* return error condition */
         }
      }

   return(pccbNew);
}
```

```
/************************* START OF SPECIFICATIONS ***************/
/*                                                               */
/* SUBROUTINE NAME:  ioFindCodec                                 */
/*                                                               */
/* DESCRIPTIVE NAME:                                             */
/*                                                               */
/* FUNCTION: This function finds a compression/decompression     */
/* control block for this compression type.                      */
/*                                                               */
/* NOTES: None                                                   */
/*                                                               */
/* ENTRY POINT: ioFindCodec                                      */
/*   LINKAGE:   CALL FAR (00:32)                                 */
/*                                                               */
/* INPUT:                                                        */
/*           PINSTANCE   pinstance  - Movie instance structure   */
/*           ULONG       ulCompressType - Compression type       */
/*                                                               */
/*                                                               */
/* EXIT-NORMAL:                                                  */
/*              pccb                                             */
/*                                                               */
/* EXIT-ERROR:                                                   */
/*              NULL - 0L                                        */
/*                                                               */
/*                                                               */
/* SIDE EFFECTS:                                                 */
/*                                                               */
/************************* END OF SPECIFICATIONS *****************/
PCCB ioFindCodec ( PINSTANCE pinstance,
                   ULONG ulCompressType )

{
   PCCB   pccb;

   for (pccb = pinstance->pccbList; pccb; pccb = pccb->pccbNext) {
      if (pccb->cifi.ulCompressType == ulCompressType)
         return(pccb);
      }
   return(NULL);  /* not found */
}
```

The ffOpenCodec routine shown in the following example opens a CODEC instance for a movie instance and creates file format specific header information (including a source video header and a destination video header). Next, it calls MMIO services to open the CODEC.

```
LONG ffOpenCodec ( PINSTANCE pinstance,
                   PCCB pccb,
                   ULONG hCodec,
                   PTRACKI ptracki)

{
   LONG              rc = MMIO_SUCCESS;/* Return code of IOProc's call. */
   PMMVIDEOHEADER     pmmVideoHdr;     /* Video header node.      */
   PCODECVIDEOHEADER  pcodecvidhdr;

   /*************************************************************/
   /* Fill in necessary information for DC IOProc.              */
   /*************************************************************/
        /* Get standard track header */
   pmmVideoHdr = (PMMVIDEOHEADER)ptracki->pTrackHeader;
   pccb->codecopen.pControlHdr = NULL;
     // FOR AVI, fill in from CODEC SPECIFIC DATA SECTION OF HEADER!
   pccb->codecopen.pOtherInfo = NULL;

   ENTERCRITX;
   /*****************************************/
   /* Create Source Video Header            */
   /*****************************************/
   if (pcodecvidhdr = (PCODECVIDEOHEADER)HhpAllocMem(hheap,(ULONG)
      sizeof(CODECVIDEOHEADER))) {
      pccb->codecopen.pSrcHdr = (PVOID)pcodecvidhdr;

      pcodecvidhdr->ulStructLen = sizeof(CODECVIDEOHEADER);
      pcodecvidhdr->cx = pmmVideoHdr->ulWidth;
      pcodecvidhdr->cy = pmmVideoHdr->ulHeight;
```

```
      pcodecvidhdr->cPlanes = 1;                       /* Hardcoded  */
      pcodecvidhdr->cBitCount = 16;                    /* Hardcoded  */
      pcodecvidhdr->ulColorEncoding = MMIO_COMPRESSED;  /* Hardcoded  */

      /*****************************************/
      /* Create Destination Video Header          */
      /*****************************************/
      if (pcodecvidhdr = (PCODECVIDEOHEADER)HhpAllocMem(hheap,(ULONG)
         sizeof(CODECVIDEOHEADER))) {
         pccb->codecopen.pDstHdr = (PVOID)pcodecvidhdr;

         pcodecvidhdr->ulStructLen = sizeof(CODECVIDEOHEADER);
         pcodecvidhdr->cx = pmmVideoHdr->ulWidth;
         pcodecvidhdr->cy = pmmVideoHdr->ulHeight;
         pcodecvidhdr->cPlanes = 1;                        // Hardcoded

         /*****************************************/
         /* Initialize the Flags and color encoding  */
         /*****************************************/
         pccb->codecopen.ulFlags = pccb->cifi.ulCapsFlags &
               (VALID_CODECOPEN_INPUTFLAGS);

         /* Set the color depth for the CODEC we want */
         if (ulNumColors == 16) {
            pccb->codecopen.ulFlags |= CODEC_4_BIT_COLOR;
            pcodecvidhdr->cBitCount = 16;
            pcodecvidhdr->ulColorEncoding = MMIO_PALETTIZED;
            }
         else if (ulNumColors > 256) {
            pccb->codecopen.ulFlags |= CODEC_16_BIT_COLOR;
            pcodecvidhdr->cBitCount = 256;
            pcodecvidhdr->ulColorEncoding = MMIO_RGB_5_6_5;
            }
         else { /* 256 and anything else */
            pccb->codecopen.ulFlags |= CODEC_8_BIT_COLOR;
            pcodecvidhdr->cBitCount = 8;
            pcodecvidhdr->ulColorEncoding = MMIO_PALETTIZED;
            }

         /*****************/
         /* Open the CODEC
         /*****************/
         rc = pccb->pmmioproc(&hCodec,
                              MMIOM_CODEC_OPEN,
                              (LONG)&pccb->codecopen,
                              0L);
         if (!rc) {
            pccb->hCodec = hCodec;
            }
         }
      }

   EXITCRIT;
   return(rc);
}


LONG ffAssociateCodec ( PINSTANCE pinstance,
                        PMMEXTENDINFO pmmextendinfo )
{
   LONG  rc = MMIO_SUCCESS;

   return(rc);
}
```

Compression

The following section illustrates how to compress raw digital images into a smaller form so they can use less storage space. Follow these steps for compression support:

1. Determine which CODEC procedure to use.
2. Load the CODEC DLL file.
3. Load and initialize the CODEC procedure.
4. Call the CODEC to compress the data.
5. Close the CODEC and release resources after use.

The IOSET.C file shown in the following example uses the MMEXTENDINFO data structure to set values for the source header, destination header, and other information.

Note: You can only associate one stream or track.

```
/************************START OF SPECIFICATIONS ***************/
/* SOURCE FILE NAME:  IOSET.C                                   */
/*                                                              */
/* DESCRIPTIVE NAME: File Format IOProc routine for MMIOM_SET   */
/*                                                              */
/* COPYRIGHT:     IBM - International Business Machines         */
/*            Copyright (c) IBM Corporation  1991, 1992, 1993   */
/*                         All Rights Reserved                  */
/*                                                              */
/* STATUS: OS/2 Release 2.0                                     */
/*                                                              */
/* FUNCTION: This source module contains the set functions.     */
/* NOTES:                                                       */
/*    DEPENDENCIES: none                                        */
/*    RESTRICTIONS: Runs in 32-bit protect mode (OS/2 2.0)      */
/*                                                              */
/* ENTRY POINTS:                                                */
/*              IOProcSet                                       */
/*                                                              */
/************************ END OF SPECIFICATIONS *****************/

#include        <stdio.h>
#include        <string.h>
#include        <stdlib.h>
#include        <memory.h>

#define         INCL_DOS            /* #define  INCL_DOSPROCESS.*/
#define         INCL_ERRORS
#define         INCL_WIN
#define         INCL_GPI

#include        <os2.h>             /* OS/2 headers.            */
#include        <pmbitmap.h>

#define         INCL_OS2MM
#define         INCL_MMIO_CODEC
#define         INCL_MMIO_DOSIOPROC

#include        <os2me.h>           /* Multimedia IO extensions. */
#include        <hhpheap.h>
#include        <ioi.h>

/************************* START OF SPECIFICATIONS **********/
/*                                                          */
/* SUBROUTINE NAME: IOProcSet                               */
/*                                                          */
/* DESCRIPTIVE NAME: Set various conditions in IOProc       */
/*                                                          */
/* FUNCTION:                                                */
/*                                                          */
/* NOTES: None                                              */
/*                                                          */
/* ENTRY POINT: IOProcSet                                   */
/*   LINKAGE:   CALL FAR (00:32)                            */
/*                                                          */
/* INPUT:                                                   */
/*   PMMIOINFO   pmmioinfo - ptr to instance structure      */
/*   LONG        lParam1  - first parameter                 */
/*   LONG        lParam2  - Second parameter                */
/*                                                          */
/*                                                          */
/* EXIT-NORMAL:                                             */
/*              MMIO_SUCCESS                                */
/*                                                          */
```

```
/* EXIT-ERROR:                                                  */
/*              MMIO_ERROR                                      */
/*                                                              */
/* SIDE EFFECTS:                                                */
/*                                                              */
/************************** END OF SPECIFICATIONS ***********/
LONG IOProcSet ( PMMIOINFO pmmioinfo,
                 LONG lParam1,
                 LONG lParam2 )

{
   PINSTANCE       pinstance;
   LONG            rc = MMIO_SUCCESS;
   PMMEXTENDINFO   pmmextendinfo = (PMMEXTENDINFO)lParam1;
   PCCB            pccb;
   ULONG           ulCCBCount;
   PCODECASSOC     pcodecassoc;
   ULONG           ulSize;
   PVOID           PtrNextAvail;

   if (rc = ioGetPtrInstance(pmmioinfo,&pinstance))
      return(rc);

   switch(lParam2){
      /************************/
      /* SET INFO             */
      /************************/
      case MMIO_SET_EXTENDEDINFO:   /* Set extended information */
         if (pmmextendinfo) {     /* error check */

            /********************/
            /* Set active track */
            /********************/
            if (pmmextendinfo->ulFlags & MMIO_TRACK) {

               if (pmmextendinfo->ulTrackID == (ULONG)MMIO_RESETTRACKS) {
                  pinstance->lCurrentTrack = pmmextendinfo->ulTrackID;
                  }
               else {
                 if (pinstance->ulFlags & OPENED_READONLY) {
                   if (ioFindTracki(pinstance,pmmextendinfo->ulTrackID)) {
                      pinstance->lCurrentTrack = pmmextendinfo->ulTrackID;
                       }
                     else {
                        pmmioinfo->ulErrorRet = MMIOERR_INVALID_PARAMETER;
                        rc = MMIO_ERROR;
                        break;
                        }
                     }

                  else if (pinstance->ulFlags &
                          (OPENED_READWRITE | OPENED_WRITECREATE)) {
                     pinstance->lCurrentTrack = pmmextendinfo->ulTrackID;
                     }

                  else {
                     pmmioinfo->ulErrorRet = MMIOERR_INVALID_PARAMETER;
                     rc = MMIO_ERROR;
                     break;
                     }
                  } /* else */
               }  /* MMIO_TRACK */

            /*********************************************************/
            /* Reset all Non-normal reading modes.  All request audio */
            /* and video frames are returned.                        */
            /*********************************************************/
            if (pmmextendinfo->ulFlags & MMIO_NORMAL_READ) {
               pinstance->ulMode = MODE_NORMALREAD;
               } /* MMIO_NORMAL_READ */

            /*********************************************************/
            /* Set IOProc into SCAN mode for the active track. Reading */
            /* will now be done, but only Key frames are returned for  */
            /* video.                                                  */
            /*********************************************************/
            else if (pmmextendinfo->ulFlags & MMIO_SCAN_READ) {
               pinstance->ulMode = MODE_SCANREAD;
               } /* MMIO_SCAN_READ */
```

```
      /*********************************************************/
      /* Set IOProc into REVERSE mode for the active track.    */
      /* Reading will now be done, but only Key frames are     */
      /* returned for video                                    */
      /*********************************************************/
      else if (pmmextendinfo->ulFlags & MMIO_REVERSE_READ) {
         pinstance->ulMode = MODE_REVERSEREAD;
         } /* MMIO_REVERSE_READ */

      /*************************************************/
      /* Associate CODEC information for recording     */
      /*************************************************/
      if (pmmextendinfo->ulFlags & MMIO_CODEC_ASSOC) {

         /* Don't alow a CODEC association for read only files */
         if (pinstance->ulFlags & OPENED_READONLY) {
            pmmioinfo->ulErrorRet = MMIOERR_INVALID_PARAMETER;
            rc = MMIO_ERROR;
            break;
            }

         /* Can only associate 1 CODEC for record */
         if (pmmextendinfo->ulNumCODECs == 1) {
            if (rc = ioAssociateCodec(pmmioinfo,
                                      pinstance,
                                      pmmextendinfo->pCODECAssoc)) {
               pmmioinfo->ulErrorRet = rc;
               rc = MMIO_ERROR;
               }
            }
         else {
            pmmioinfo->ulErrorRet = MMIOERR_INVALID_PARAMETER;
            rc = MMIO_ERROR;
            }
         } /* MMIO_CODEC_ASSOC */
      } /* pmmextendedinfo */

   else { /* error - data structure missing */
      pmmioinfo->ulErrorRet = MMIOERR_INVALID_PARAMETER;
      rc = MMIO_ERROR;
      }
   break;
/*******************************/
/* QUERY BASE AND CODEC INFO    */
/*******************************/
case MMIO_QUERY_EXTENDEDINFO_ALL:
  /* Query Also CODEC associated info */

 /* Create the array of codecassoc structures to return to caller */
 pcodecassoc = pmmextendinfo->pCODECAssoc;  /* Point to beginning */
 for (pccb = pinstance->pccbList; pccb; pccb = pccb->pccbNext) {
    pcodecassoc->pCodecOpen = NULL;
    pcodecassoc->pCODECIniFileInfo = NULL;
    pcodecassoc++;
    }
  PtrNextAvail = (PVOID)pcodecassoc;

  /* Fill in pointers to the CODECIniFileInfo structures to follow */
  ulSize = 0L;
  pcodecassoc = pmmextendinfo->pCODECAssoc;  /* Point to beginning */
  for (pccb = pinstance->pccbList; pccb; pccb = pccb->pccbNext) {

    /* Create and copy CODECINIFILEINFO structure */
    pcodecassoc->pCODECIniFileInfo = (PCODECINIFILEINFO)PtrNextAvail;
    memcpy(pcodecassoc->pCODECIniFileInfo,&pccb->
       cifi,sizeof(CODECINIFILEINFO));
    PtrNextAvail = (PVOID) (((ULONG)PtrNextAvail)
       + sizeof(CODECINIFILEINFO));

     /* Create and copy CODECOPEN structure */
     pcodecassoc->pCodecOpen = PtrNextAvail;
     memcpy(pcodecassoc->pCodecOpen,&pccb->codecopen,
        sizeof(CODECOPEN));
     PtrNextAvail = (PVOID) (((ULONG)PtrNextAvail)
        + sizeof(CODECOPEN));

      /* Create and copy Pointers to structures   */
      /* in the CODECOPEN structure.              */
```

```
      if (pccb->codecopen.pControlHdr) {
         ulSize = *((PULONG)pccb->codecopen.pControlHdr);
         ((PCODECOPEN)pcodecassoc->pCodecOpen)->pControlHdr =
            (PVOID)PtrNextAvail;
         memcpy(((PCODECOPEN)pcodecassoc->pCodecOpen)->pControlHdr,
                pccb->codecopen.pControlHdr,
                ulSize);
         PtrNextAvail = (PVOID) (((ULONG)PtrNextAvail) + ulSize);
         }
      if (pccb->codecopen.pSrcHdr) {
         ulSize = *((PULONG)pccb->codecopen.pSrcHdr);
         ((PCODECOPEN)pcodecassoc->pCodecOpen)->pSrcHdr
             = PtrNextAvail;
         memcpy(((PCODECOPEN)pcodecassoc->pCodecOpen)->pSrcHdr,
                pccb->codecopen.pSrcHdr,
                ulSize);
         PtrNextAvail = (PVOID) (((ULONG)PtrNextAvail) + ulSize);
         }

      if (pccb->codecopen.pDstHdr) {
         ulSize = *((PULONG)pccb->codecopen.pDstHdr);
         ((PCODECOPEN)pcodecassoc->pCodecOpen)->pDstHdr
             = PtrNextAvail;
         memcpy(((PCODECOPEN)pcodecassoc->pCodecOpen)->pDstHdr,
                pccb->codecopen.pDstHdr,
                ulSize);
         PtrNextAvail = (PVOID) (((ULONG)PtrNextAvail) + ulSize);
         }

      if (pccb->codecopen.pOtherInfo) {
         ulSize = *((PULONG)pccb->codecopen.pOtherInfo);
         ((PCODECOPEN)pcodecassoc->pCodecOpen)->pOtherInfo
              = PtrNextAvail;
         memcpy(((PCODECOPEN)pcodecassoc->pCodecOpen)->pOtherInfo,
                pccb->codecopen.pOtherInfo,
                ulSize);
         PtrNextAvail = (PVOID) (((ULONG)PtrNextAvail) + ulSize);
         }
      pcodecassoc++;
      }

/***********************************************/
/* QUERY BASE INFO  (NOTE: Fall through    */
/* from previous case!)                        */
/***********************************************/
case MMIO_QUERY_EXTENDEDINFO_BASE: /* Query only MMEXTENDINFO info */
   pmmextendinfo->ulStructLen = sizeof(MMEXTENDINFO);
   pmmextendinfo->ulTrackID = (ULONG)pinstance->lCurrentTrack;
   /* pmmextendinfo->pCODECAssoc = NULL;  */

   /* Compute ulBufSize for complete information return */
   ulSize = 0L;
   for (pccb = pinstance->pccbList, ulCCBCount = 0; /* Count CCB's */
        pccb;
        ulCCBCount++, pccb = pccb->pccbNext) {
      ulSize += sizeof(CODECASSOC)+sizeof(CODECOPEN)
          +sizeof(CODECINIFILEINFO); /* static stuff */

      /* Extract ulStructLen as first field of structure  */
      /* that ptr points to.                              */
      if (pccb->codecopen.pControlHdr) {
         ulSize += *((PULONG)pccb->codecopen.pControlHdr);
         }
      if (pccb->codecopen.pSrcHdr) {
         ulSize += *((PULONG)pccb->codecopen.pSrcHdr);
         }
      if (pccb->codecopen.pDstHdr) {
         ulSize += *((PULONG)pccb->codecopen.pDstHdr);
         }
      if (pccb->codecopen.pOtherInfo) {
         ulSize += *((PULONG)pccb->codecopen.pOtherInfo);
         }
      }

   pmmextendinfo->ulNumCODECs = ulCCBCount;
   pmmextendinfo->ulBufSize = ulSize;
   break;

/*********/
```

```
      /* ERROR */
      /*********/
      default:
         pmmioinfo->ulErrorRet = MMIOERR_INVALID_PARAMETER;
         rc = MMIO_ERROR;
         break;
      }/* end switch */

   return(rc);
}
```

Associating a CODEC with a File

The following example illustrates how to associate a CODEC procedure with a file (track).

```
LONG ioAssociateCodec ( PMMIOINFO pmmioinfo,
                        PINSTANCE pinstance,
                        PCODECASSOC pcodecassoc )

{
   LONG              rc = MMIO_SUCCESS; /* Return code of IOProc's call.*/
   PCCB              pccb;
   ULONG             hCodec;  /* Possibly returned from ioLoadCodecDLL */

   /* Check for NULL pointers */
   if (!pcodecassoc->pCodecOpen || !pcodecassoc->pCODECIniFileInfo) {
      return (MMIOERR_INVALID_PARAMETER);
      }

   /* Force the correct values into the CODECINIFILEINFO structure */
   pcodecassoc->pCODECIniFileInfo->ulStructLen = sizeof(CODECINIFILEINFO);
   pcodecassoc->pCODECIniFileInfo->fcc = pmmioinfo->fccIOProc;
   pcodecassoc->pCODECIniFileInfo->ulCapsFlags |= CODEC_COMPRESS;

   /* Find the codec to load */
   if (rc = ioDetermineCodec(pinstance, 0, pcodecassoc->pCODECIniFileInfo))
      {
      return(rc); /* return error */
      }

   else { /* load and open the compression CODEC */

      /*********************************************/
      /* Check for previously installed CODECs.    */
      /* De-install any loaded.  Load new one.     */
      /* Allows only 1 CODEC to be loaded at a time. */
      /*********************************************/
      if (pinstance->pccbList) {
         pccb = pinstance->pccbList;
         pinstance->pccbList = pccb->pccbNext;   /* unlink from list */
         ioCloseCodec(pccb);
         }

      /* Load the CODEC DLL */
      if (pccb = ioLoadCodecDLL(pinstance,
                                pcodecassoc->pCODECIniFileInfo,
                                &hCodec)) {

         /* Save the CODEC open information in the ccb */
         ((PCODECOPEN)pcodecassoc->pCodecOpen)->ulFlags |= CODEC_COMPRESS;
         /* Force open of compressor */

         if (!(rc = ioInitCodecopen(pccb,(PCODECOPEN)pcodecassoc->
            pCodecOpen))) {

            /* Open the CODEC */
            if (!(rc = pccb->pmmioproc(&hCodec,
                                       MMIOM_CODEC_OPEN,
                                       (LONG)&pccb->codecopen,
```

```
                                    0L)))  {
               pccb->hCodec = hCodec;       /* save handle to CODEC */
               }
            }

         /* handle error conditions */
         if (rc) {
            pinstance->pccbList = pccb->pccbNext;   /* unlink from list */
            ioCloseCodec(pccb);
            }
         }
      else {
         rc = MMIO_ERROR;
         }
      }
   return(rc);
}
```

Allocating Memory for Compression

The ioAssociateCodec routine shown in the following example calls the ioInitCodecopen routine to allocate memory. The next example illustrates how the ioInitCodecopen routine allocates memory and initializes a CODECOPEN structure to be saved in the CODEC control block (CCB).

```
LONG ioInitCodecopen ( PCCB pccb,
                       PCODECOPEN pcodecopen)

{
   ULONG             ulSize;


   ENTERCRITX;
   pccb->codecopen.ulFlags = pcodecopen->ulFlags;

   /* Create and copy Pointers to structures in CODECOPEN structure */
   if (pcodecopen->pControlHdr) {
     ulSize = *((PULONG)pcodecopen->pControlHdr);
     if (!(pccb->codecopen.pControlHdr = (PVOID)HhpAllocMem(hheap,ulSize)))
        {
         return(MMIO_ERROR);
         }
      memcpy(pccb->codecopen.pControlHdr, pcodecopen->pControlHdr, ulSize);
      }

   if (pcodecopen->pSrcHdr) {
      ulSize = *((PULONG)pcodecopen->pSrcHdr);
      if (!(pccb->codecopen.pSrcHdr = (PVOID)HhpAllocMem(hheap,ulSize)))
         {
         return(MMIO_ERROR);
         }
      memcpy(pccb->codecopen.pSrcHdr, pcodecopen->pSrcHdr, ulSize);
      }

   if (pcodecopen->pDstHdr) {
      ulSize = *((PULONG)pcodecopen->pDstHdr);
      if (!(pccb->codecopen.pDstHdr = (PVOID)HhpAllocMem(hheap,ulSize)))
         {
         return(MMIO_ERROR);
         }
      memcpy(pccb->codecopen.pDstHdr, pcodecopen->pDstHdr, ulSize);
      }

   if (pcodecopen->pOtherInfo) {
      ulSize = *((PULONG)pcodecopen->pOtherInfo);
      if (!(pccb->codecopen.pOtherInfo = (PVOID)HhpAllocMem(hheap,ulSize)))
      {
         return(MMIO_ERROR);
         }
      memcpy(pccb->codecopen.pOtherInfo, pcodecopen->pOtherInfo, ulSize);
```

```
      }

   EXITCRIT;
   return(MMIO_SUCCESS);
}
```

Closing the CODEC Procedure

The following example shows an example of how to close a CODEC instance. The ioCloseCodec routine frees memory associated with the CODEC.

```
LONG ioCloseCodec ( PCCB pccb )

{
   LONG       rc = MMIO_SUCCESS; /* Return code of IOProc's call. */

   ENTERCRITX;
   if (pccb->codecopen.pSrcHdr) {
      HhpFreeMem(hheap,(PVOID)pccb->codecopen.pSrcHdr);
      }

   if (pccb->codecopen.pDstHdr) {
      HhpFreeMem(hheap,(PVOID)pccb->codecopen.pDstHdr);
      }

   if (pccb->codecopen.pControlHdr) {
      HhpFreeMem(hheap,(PVOID)pccb->codecopen.pControlHdr);
      }

   if (pccb->codecopen.pOtherInfo) {
      HhpFreeMem(hheap,(PVOID)pccb->codecopen.pOtherInfo);
      }

   if (pccb->hCodec) {
      rc = pccb->pmmioproc(&pccb->hCodec,
                           MMIOM_CODEC_CLOSE,
                           0L,
                           0L);

      if (!rc) {
         pccb->hCodec = 0L;
         }
      }

   if (pccb->hmodule) {
//----DosFreeModule(pccb->hmodule);
      pccb->hmodule = 0;
      }

   HhpFreeMem(hheap,(PVOID)pccb);
   pccb = NULL;
   EXITCRIT;

   return(rc);
}
```

Installation Requirements

This section describes how to install the following OS/2 multimedia subsystems using the multimedia installation program (MINSTALL):

- Media control driver (MCD)

- Stream handler
- I/O procedure

You can prepare script control files and if necessary, create an installation DLL file to install a subsystem. MINSTALL updates the appropriate INI files and CONFIG.SYS statements, providing a consistent installation process.

Most developers will use control files with the MINSTALL program to install a subsystem. MINSTALL is hardware independent and does not prompt the operating system for hardware information. Therefore, if information is required from the operating system or specific hardware, you need to write an installation DLL to use (in addition to control files). See Writing an Installation DLL for further information.

Master Control File

The MINSTALL program (MINSTALL.EXE) requires specific file information to install each subsystem. This file information is provided by the master control file CONTROL.SCR. The master control file, CONTROL.SCR, tells the installation program what to install, where to install it, how to display it to the user, and what system files need to be updated. CONTROL.SCR uses keywords to specify these instructions to MINSTALL.

This control file must be named CONTROL.SCR and must reside on the first media unit (diskette or CD) of the installation package. During installation, if any errors are detected in the master control file, the errors are logged in the MINSTALL.LOG file. (See Installation LOG File.)

The CONTROL.SCR file consists of a header section and a subsystem definition section. The header section comes first and provides general information for the installation procedure. The subsystem definition section comes next and provides information about the subsystems in the installation package.

CONTROL.SCR Header

The header section of the CONTROL.SCR file contains the following information:

- Name of the installation package
- Code page used when creating the file
- Name of the file list control file
- Number of subsystems in the installation package
- Number of media units required for installation
- Names of the media units required for installation
- Source and destination directory names (optional)

The following is an example of the CONTROL.SCR header located in the \TOOLKIT\SAMPLES\MM\CF subdirectory.

```
/*  control.scr */

package   ="IBM Multimedia Presentation Manager Toolkit/2"
codepage  =437
filelist  ="filelist.tlk"
groupcount=12
munitcount=6

medianame="IBM Multimedia Presentation Manager Toolkit/2 Installation
Diskette 1"
medianame="IBM Multimedia Presentation Manager Toolkit/2 Installation
Diskette 2"
.
.
.

sourcedir="\\"                    = 0
sourcedir="\\lib\\"               = 1
.
.
.

destindir="\\mmos2\\"                              = 0
```

```
destindir="\\mmos2\\mmtoolkt\\lib\\"                = 1
.
.
.
```

The following table describes the keywords used in the CONTROL.SCR header.

Keyword	Description
PACKAGE	This required keyword specifies the name of the installation package in the form of a quoted string. For example: PACKAGE="IBM Multimedia Presentation Manager Toolkit/2"
CODEPAGE	This required keyword specifies the code page that the file was created under. For example: CODEPAGE = 437
FILELIST	This required keyword specifies the name of the file list control file. This control file contains a list of files that make up each feature, identifies on which media units they reside in the installation package, and specifies the destination to which they will be copied. For example: FILELIST = "FILELIST.TLK"
GROUPCOUNT	This required keyword specifies the number of subsystems in the installation package. All groups are counted, including group 0 (if present). For example: GROUPCOUNT = 10
MUNITCOUNT	This required keyword specifies the number of media units (diskettes, CDs) that will be used if all subsystems are installed. This number must be greater than 0. This is the number of diskettes or CDs on which the installation package resides. For example: MUNITCOUNT = 6
MEDIANAME	This required keyword specifies a unique media name, which is a character string on the diskette or CD label. For each media unit, this keyword must be repeated once, in the form of a quoted string. This information is used during installation to prompt the user to insert a diskette or CD when needed. For example: MEDIANAME = "IBM Multimedia Presentation Manager Toolkit/2 Installation Diskette 1"
SOURCEDIR	This optional keyword specifies the name of a source directory and its associated number. This keyword can be repeated and is specified by a quoted string followed by an equal sign (=) and a number. The number is used to identify the particular directory in later scripts. This can be NULL, in which case two default backslash characters (\\) are used with an encoding of 0. The path must be surrounded by path separators. For example: SOURCEDIR="\\LIB\\" = 1
DESTINDIR	This optional keyword specifies the name of a destination directory and its associated number. This keyword can be repeated and is specified by a quoted string followed by an equal sign (=) and a number. The number is used to identify the particular directory in later scripts. This can be NULL, in which case two default backslash characters (\\)

```
are used with an encoding of 0.  The path
must be surrounded by path separators (\\).
For example: For example:
DESTINDIR = "\\MMOS2\\" = 0
```

Observe the following guidelines when you create or change a CONTROL.SCR header:

- You must place the keywords MUNITCOUNT and MEDIANAME so that MUNITCOUNT comes *directly* before MEDIANAME. The order of the other keywords is not significant.
- Each destination directory (DESTINDIR) must have a unique number.
- A subsystem group can be spread across several media units. It does not have to reside on one media unit.
- You can define any directory. MINSTALL will create any subdirectories defined with the DESTINDIR keyword that do not exist.
- If you move the installation package files to media of a different size, the number of media units (MEDIACOUNT) may change.
- You may use comments in the header section in the form of blank lines or text enclosed with /* and */. You may not use nested comments.
- You may use blank spaces around the equal sign; blank spaces are ignored.
- If you want to use a double quotation mark or a backslash in a string, you must precede it with the escape character (\).
- You may use the escape sequence \n (new line).

CONTROL.SCR Subsystem Definition

The subsystem definition section of the CONTROL.SCR file follows the header section and contains the definitions for each of the subsystems in the installation package. A block of information must be included for each feature.

Each block of information in the subsystem definition section contains the following information:

- The group or feature number
- The feature name
- The version of the component
- The size of all the files for the feature installation
- The names of the control files that change the INI files and CONFIG.SYS statements
- The names of DLL files and entry points

The following is an example of a CONTROL.SCR subsystem definition.

```
ssgroup  =0
ssname   ="Toolkit_Base"
ssversion="0.63.0"
sssize   =13

/*           - 7  = clock    */
ssgroup  =7
ssname   ="Clock Utility"
ssversion="0.63.0"
sssize   =839
ssinich  ="TLKCLOCK.SCR"

/*           - 8  = midiconv  */
ssgroup  =8
ssname   ="MIDI File Format Converter"
ssversion="0.63.0"
sssize   =170
ssinich  ="TLKCONV.SCR"

/*           - 3  = midiio   */
ssgroup  =3
ssname   ="MIDI IO Procedure"
ssversion="0.63.0"
```

```
sssize   =475

/*           - 11 = mcistrng */
ssgroup  =11
ssname   ="Media Control Interface String Test"
ssversion="0.63.0"
sssize   =194
ssinich  ="TLKSTRN.SCR"

/*           - 9  = duet1   */
ssgroup  =9
ssname   ="Duet Player I"
ssversion="0.63.0"
sssize   =4854
ssinich  ="TLKDUT1.SCR"

/*           - 10 = duet2     */
ssgroup  =10
ssname   ="Duet Player II"
ssversion="0.63.0"
sssize   =816
ssinich  ="TLKDUT2.SCR"

/*           - 12 = inc, lib, and h   */
ssgroup  =12
ssname   ="Header Files, Include Files and Libraries"
ssversion="0.63.0"
sssize   =384
ssconfigch="TOOLKIT.CH"

/*           - 13 = prog ref   */
ssgroup   =13
ssname   ="Program Reference"
ssversion="0.63.0"
sssize   =400
ssinich  ="TLKBOOKR.SCR"

/*           - 14 = workbook   */
ssgroup  =14
ssname   ="Workbook"
ssversion="0.63.0"
sssize   =150
ssinich  ="TLKBOOKW.SCR"

/*           - 2 = avcinst   */
ssgroup  =2
ssname   ="AVC I/O Procedure Installation Utility"
ssversion="0.63.0"
sssize   =102
ssinich  ="TLKIOPU.SCR"

/*           - 4 = caseconv   */
ssgroup  =4
ssname   ="Case Converter I/O Procedure"
ssversion="0.63.0"
sssize   =82
```

The CONTROL.SCR subsystem definition consists of the keywords shown in the folloiwng table.

Keyword	Description
SSGROUP	This required keyword specifies the group. This marks the beginning of a group and assigns it a number. Each group must have a unique number from 0-49 within the package; however, the same number can be used with different installation packages. The groups are displayed in the installation main selection window in ascending order by group number. For example: SSGROUP = 5
SSNAME	This required keyword specifies the group name, as an ASCII string. This keyword is case sensitive and takes the

form of a quoted string. The name may include special characters and may be translated. The name is displayed in the main installation selection window. For example: SSNAME = "CD Audio"

SSVERSION — This required keyword specifies the version of the group in the form of a quoted string. This string must be in the format "*dd.dd.dd*" (where *dd* represents digits). Any version not specified in this format will be converted to that format. All string items that are not digits or periods will be converted to zeros. Any periods after the second period will be converted to zeros. For example: SSVERSION = "1.1.0"

SSSIZE — This required keyword specifies the total size of all the files in the group. The size denotes the number of bytes in thousands (500 = 500KB). This number is used to help determine if there is enough disk space to support the installation. If you do not know the correct size of a group, overstate its size. For example: SSSIZE = 1024

SSINICH — This optional keyword specifies the name of the file that contains changes to an INI file. If this statement is missing, there are no changes to an INI file. For example: SSINICH = "ACPAINI.CH"

SSCONFIGCH — This optional keyword specifies the name of the file that contains the changes to the CONFIG.SYS file. If this statement is missing, there are no changes to the CONFIG.SYS file. For example: SSCONFIGCH = "ACPACON.CH"

SSCOREQS — This optional keyword specifies a list of corequisites needed for this group to run. It specifies what other groups the current group depends on. These other groups must be installed for the current group to function. (If this statement is missing, there are no corequisites.) The corequisite is identified by its group number. Corequisite groups should point to each other only if they require each other. It is possible to have group A list group B as a corequisite and group B have no corequisites. If the user selects a group with corequisites, but does not select all the corequisites, the user is notified before the installation starts. This entry can be repeated as necessary. For example: SSCOREQS = 1

SSICON — This optional keyword names the icon file for this group that is to be displayed in the installation main selection window. The icon file to be displayed in the selection window must reside on the first installation media unit. If this statement is missing, a default icon is used. For example: SSICON = "ACPA.ICO"

SSDLL — This optional keyword names a DLL file that is to be run during the installation process. The DLL referenced will be run after all files

	are copied to the destination, but before any script processing is performed. If this keyword is present, the SSDLLENTRY keyword must also be present. For example: SSDLL="MY.DLL"
SSDLLENTRY	This optional keyword specifies the name of the entry point into SSDLL in the form of a quoted string. If this keyword is present, the SSDLL keyword must also be present. For example: SSDLLENTRY="MyEntry"
SSTERMDLL	This optional keyword names a DLL file that is to be run during the installation process. The DLL referenced will be run after all files are copied to the destination and after all script processing is done. The purpose of this keyword is to allow for processing to occur on a fully configured multimedia system. If this keyword is present, the SSTERMDLLENTRY keyword must also be present. For example: SSTERMDLL="MYTERM.DLL"
SSTERMDLLENTRY	This optional keyword specifies the name of the entry point into SSTERMDLL in the form of a quoted string. If this keyword is present, the SSTERMDLL keyword must also be present. For example: SSTERMDLLENTRY="MyTermEntry"
SSDLLINPUTPARMS	This optional keyword specifies information needed by an installation DLL in the form of a quoted string. This information is passed as a parameter to the installation DLL as specified in the SSDLL or SSTERMDLL keywords. For example: SSDLLINPUTPARMS="5, FILE.NAM, 1.1.0"
SSSELECT	This optional keyword determines the preselection of subsystems for installation. Five values are supported: "ALWAYS" - This value specifies that the group will always be installed. It is the only valid value for group 0. Groups with this value will not be displayed in the installation main selection window. "REQUIRED" - This value specifies that the group will be preselected for installation. If the group had been previously installed, it cannot be unselected by the user if this installation package is newer than the installed version. If the group had not been previously installed, it can be unselected by the user. "VERSION" - This value specifies that the group will be preselected only if it was previously installed and this installation package is newer than the installed version. However, it can be unselected by the user. "YES" - This value specifies that the group will be preselected whether or not it was previously installed. It can be unselected by the user. This is the default. "NO" - This value specifies that the group is never preselected but can be selected by the user. "BASENEWER" - This value specifies that

```
                    files belonging to this group will only
                    be copied if the user's machine has no
                    package installed or if the package
                    installed is older than the current
                    package.
                    "ONLYNEWER" - This value specifies that
                    a user will not be able to install an
                    older version of a package on top of a
                    newer version. Files belonging to this
                    group will only be copied if the user
                    has an older version (or the same
                    version) installed.  If no version is
                    installed or if the version installed is
                    higher than the one in the package, no
                    files will be copied.
```

Observe the following guidelines when you create or change a CONTROL.SCR subsystem definition:

- The SSGROUP keyword must be the first statement in the information block.
- A group may reside on different media.
- Each statement in the information block must have a value.
- You may use comments in an information block in the form of blank lines or text enclosed with /* and */. You may not use nested comments.
- You may use blank spaces around the equal sign; blank spaces are ignored.
- If you want to use a double quotation mark or a backslash in a string, you must precede it with the escape character (\).
- You may use the escape sequence \n (new line).

File List Control File

The master control file, CONTROL.SCR, specifies a FILELIST keyword which identifies the name of a file list control file that lists all the installable files in the installation package. The file list control file also contains the following:

- The name of the file
- The number of the media unit where the file is stored
- The destination directory where the file will be copied
- The group the file is identified with

The following is an example of a file list control file. (Ellipsis points indicate additional entries.) The first nonblank, noncomment record is a count of the number of files (or file name lines) in the file. For example, 145.

```
/*********************************************************************/
/* This file contains install information. Strings enclosed in C type  */
/* comments like this line are comment lines. Blank lines are ignored. */
/* Non-blank lines will be parsed and extraneous characters will       */
/* cause errors.  First non-comment line must be the total number of   */
/* files to be installed.                                              */
/*                                                                     */
/* Total number of entries is 145                                      */
/*********************************************************************/
    145
/*********************************************************************/
/* All files on the install disks are listed below.  Other information */
/* is also given, as follows:                                          */
/*                                                                     */
/* Disk#    - The number of the disk on which the file resides.        */
/*            (Ignored if installing from CD-ROM). These are sorted    */
/*            from 0 to the number of disks, ascending.                */
/*                                                                     */
/* Group#   - The logical group to which the file belongs. Group       */
/*            starts at 0.                                             */
/*                                                                     */
/* Dest#    - The destination subdirectory into which the file will be */
/*            copied.  Dest# starts at 0.                              */
/*                                                                     */
/* Source#  - The source subdirectory from which the file will be      */
/*            copied.                                                  */
/*                                                                     */
```

```
/* FileName - The base filename.                                        */
/* sourcedir="\\"                    = 0                                */
/* sourcedir="\\lib\\"               = 1                                */
/* sourcedir="\\h\\"                 = 2                                */
/* sourcedir="\\clock\\"             = 7                                */
/* sourcedir="\\duet1\\"             = 9                                */
/* sourcedir="\\duet2\\"             = 10                               */
/* sourcedir="\\mcistrng\\"          = 11                               */
/* sourcedir="\\inc\\"               = 13                               */
/* sourcedir="\\book\\"              = 16                               */
/* sourcedir="\\cdmct\\"             = 17                               */
/* sourcedir="\\avcinst\\"           = 18                               */
/* sourcedir="\\caseconv\\"          = 19                               */
/*                                                                      */
/* destindir="\\mmos2\\"                                 = 0            */
/* destindir="\\mmos2\\mmtoolkt\\lib\\"                  = 1            */
/* destindir="\\mmos2\\mmtoolkt\\h\\"                    = 2            */
/* destindir="\\mmos2\\install\\"                        = 4            */
/* destindir="\\mmos2\\mmtoolkt\\samples\\clock\\"       = 7            */
/* destindir="\\mmos2\\mmtoolkt\\samples\\duet1\\"       = 9            */
/* destindir="\\mmos2\\mmtoolkt\\samples\\duet2\\"       = 10           */
/* destindir="\\mmos2\\mmtoolkt\\samples\\mcistrng\\"    = 11           */
/* destindir="\\mmos2\\mmtoolkt\\samples\\cf\\"          = 12           */
/* destindir="\\mmos2\\mmtoolkt\\inc\\"                  = 13           */
/* destindir="\\mmos2\\dll\\"                            = 14           */
/* destindir="\\mmos2\\help\\"                           = 15           */
/* destindir="\\mmos2\\mmtoolkt\\samples\\book\\"        = 16           */
/* destindir="\\mmos2\\mmtoolkt\\samples\\cdmct\\"       = 17           */
/* destindir="\\mmos2\\mmtoolkt\\samples\\avcinst\\"     = 18           */
/* destindir="\\mmos2\\mmtoolkt\\samples\\caseconv\\"    = 19           */
/*                                                                      */
/*        groups                                                        */
/*            - 2  = AVC I/O Procedure Installation Utility             */
/*            - 3  = MIDI IO Procedure                                  */
/*            - 4  = Case Converter I/O Procedure                       */
/*            - 7  = Clock Utility                                      */
/*            - 9  = Duet Player I                                      */
/*            - 10 = Duet Player II                                     */
/*            - 11 = MCI String Test                                    */
/*            - 12 = Header Files, Include Files and Libraries          */
/*            - 13 = Program Reference                                  */
/*            - 14 = Workbook                                           */
/*                                                                      */
/* Disk# Group# Dest# Source#  FileName                                 */
/************************************************************************/

/*           mmtoolkt\samples\cf   9   14K                              */
    0    0   12    0    "CONTROL.SCR"
    0    0   12    0    "FILELIST.TLK"
    0    0   12    0    "TLKCLOCK.SCR"
    0    0   12    0    "TLKIOPU.SCR"
    0    0   12    0    "TLKCONV.SCR"
.
.
.

    0    0   4     0    "TLKCLOCK.SCR"
    0    0   4     0    "TLKIOPU.SCR"
    0    0   4     0    "TLKCONV.SCR"
    0    0   4     0    "TLKSTRN.SCR"
.
.
.

    0   10   12    0    "TOOLKIT.CH"

/*           mmtoolkt\inc     10   83K                                 */
    0   12   13   13    "ACB.INC"
    0   12   13   13    "AUDIO.INC"
    0   12   13   13    "DCB.INC"
.
.
.

/*           mmtoolkt\lib     11   50K                                 */
    0   12    1    1    "AUDCTL.LIB"
    0   12    1    1    "DSPMGRDL.LIB"
    0   12    1    1    "LVDP8000.LIB"
.
```

```
.
.

/*           mmtoolkt\h         22  269K                              */
    0   12    2    2    "ACB.H"
    0   12    2    2    "AUDCTL.H"
    0   12    2    2    "AUDIO.H"
.
.
.

/*           mmtoolkt\samples\duet2       12    390k                  */
    0   10   10   10    "duet2.c"
    0   10   10   10    "duet2.h"
    0   10   10   10    "duet2.rc"
.
.
.

/*           mmtoolkt\samples\avcinst   9   102K                      */
    0    2   18   18    "avcinst.dlg"
    0    2   18   18    "avcinst.def"
    0    2   18   18    "avcinst.ipf"
.
.
.

/*           mmtoolkt\samples\caseconv  8    72K                      */
    0    4   19   19    "convcvsr.c"
    0    4   19   19    "convproc.rc"
    0    4   19   19    "convconv.c"
.
.
.

/*           mmtoolkt\samples\midiconv  2    44K                      */
    1    8   15    0    "midiconv.hlp"
    1    8    0    0    "midiconv.exe"

/*           mmtoolkt\samples\mcistrng  14   194K                     */
    1   11   11   11    "mcistrng.c"
    1   11   11   11    "mcistrng.h"
    1   11   11   11    "mcistrng.rc"
.
.
.
```

The following table describes the columns in the file list control file.

Column	Description
Media#	Specifies the number of the media unit (diskette or CD) where the file is stored. The units are numbered starting from 0. This number will be used for all installation media except for the hard disk. The Media# column must be sorted in ascending order. A media unit does not have to be filled (there can be unused space on any numbered unit).
Group or subsystem#	Specifies the group to which the file belongs. The group number must be a positive integer, with numbering starting at 0 (the groups are defined in CONTROL.SCR by the SSGROUP keyword). This number is used to determine which files belong to a group selected for installation.
Destination#	Specifies the destination subdirectory where the file will be copied. The encoding mapping is defined in the CONTROL.SCR file by the DESTINDIR keyword. This field must always be a defined number (for example, 14 for the \MMOS2\DLL path). If you specify a DESTINDIR statement in the master control file, you only have to specify

```
                the corresponding group number (for example,
                1).

 Source#        Specifies the path name of the source file.
                The encoding mapping is defined in the
                CONTROL.SCR file by the SOURCEDIR keyword.
                This field must always be defined with a
                number (for example, 1 for the \LIB path).

 File name      Specifies the source file name, which must be
                in double quotes.  For example,
                "MINSTALL.EXE".
```

Change Control Files

Change control files are script files that make changes to the CONFIG.SYS and INI files. The master control file, CONTROL.SCR, identifies the change control files when you specify the SSCONFIGCH and SSINICH keywords.

Supported Macros

Macros can be used in the change control files. Macros can also be used in the master control file. When a supported macro is used, drives and paths do not have to be identified until the time of installation. At installation, macros can perform the following:

- Replace the full destination path of the file
- Replace the installation target drive letter
- Replace the default destination drive and path as defined in CONTROL.SCR
- Replace the startup (boot) drive letter of the operating system
- Delete specified files

The following describes the supported macros.

Macro	**Description**
destindir= "$(DELETE)*path* \\" = *number*	The $(DELETE) macro can only be used with the DESTINDIR keyword in the master control file. The relative *number* is listed in the file list control file. Every file that has this number will be deleted from the user's machine.

$(DEST)*filename*	$(DEST) is replaced with the full destination path of the file. For example:
	`$(DEST)CLOCK.EXE`
	becomes
	`D:\MMOS2\MMTOOLKT\SAMPLES\CLOCK\CLOCK.EXE`

$(DRIVE)	$(DRIVE) is replaced with the installation target drive letter. (Do not append a colon.) For example: `$(DRIVE)\MMOS2\TEST1.EXE` becomes

	`D:\MMOS2\TEST1.EXE`

$(DIR)#	*#* is the number of the destination directory as stated in the CONTROL.SCR file. The macro is replaced with the drive and path defined in the CONTROL.SCR file for the specified DESTINDIR definition. For example:
	`$(DIR)4\MINSTALL.LOG`
	becomes
	`D:\MMOS2\INSTALL\MINSTALL.LOG`

$(BOOT)	Replaces the startup (boot) drive letter of the operating system. (Do not append a colon.) For example:
	`$(BOOT)\OS2\NEW.SYS`
	becomes
	`C:\OS2\NEW.SYS`
	where C: is the drive on which OS/2 is installed.

Note: Using multiple macros is supported.

CONFIG.SYS Change Control Files

Some multimedia subsystems require unique statements in the CONFIG.SYS file. The CONFIG.SYS change control file creates, adds, merges, and replaces CONFIG.SYS statements through the use of keywords. The master control file, CONTROL.SCR, specifies a SSCONFIGCH keyword for each file that contains changes to the CONFIG.SYS file.

The following describes the keywords used in the CONFIG.SYS file.

Keyword	**Description**

DEVICE	Adds new device statements to the CONFIG.SYS file. The right side of the equal sign must be a quoted string. For example:
	`DEVICE="$(DEST)DEVICE.SYS /parameters"`
	A supported macro may be used. Ordinarily, the $(DEST) macro is used. When the $(DEST) macro is used, MINSTALL searches for the file named DEVICE.SYS in the control files. If MINSTALL finds it, the $(DEST) macro is replaced with the full path of the destination of that file, and the final statement is added to the CONFIG.SYS file. For example:
	`DEVICE=d:\SOMEDIR\DEVICE.SYS /parameters`

MERGE	Merges data into an existing statement in the CONFIG.SYS file. For example: `MERGE "LIBPATH"=1` The number 1 relates to the CONTROL.SCR file destination subdirectory 1. After MINSTALL finds subdirectory 1, it adds this path at the end of the current LIBPATH statement. (A semicolon ends a line and is used between components.) If there is no current LIBPATH statement in the file, a LIBPATH statement will be generated with the specified path.

If the right side of the equal sign is not a numeric string, the line is copied as is. For example:

```
MERGE "SOMEVAR"="WHOKNOWS"
```

If the SOMEVAR environment variable exists, WHOKNOWS is added at the end of the statement. If the variable does not exist, the statement:

```
SOMEVAR=WHOKNOWS
```

is added to the CONFIG.SYS file.

Note: Refer to the TOOLKIT.CH file in the \TOOLKIT\SAMPLES\MM\CF subdirectory.

You also can insert SET following the first quotation mark character inside the first quoted string. The word SET is ignored unless you are adding a new line to the CONFIG.SYS file. In this case, SET is appended to the beginning of the line. For example:

```
MERGE "SET SOMEVAR"="WHOKNOWS"
```

REPLACE

Replaces an existing CONFIG.SYS statement.

REPLACE can be followed by a variable name or a variable name preceded by SET inside a quoted string. On the right side of the equal sign is either a number or a quoted string. A supported macro may be used. For example:

```
REPLACE "SET MMBASE" = 0
```

If the variable MMBASE is in the CONFIG.SYS file, then the value is changed to 0. If the variable does not exist, the following statement is added:

```
SET MMBASE=C:\MMOS2;
```

Note: This statement replaces the existing statement. Do not use this statement in conjunction with a path statement.

INI Change Control Files

You can use an INI change control file to do the following:

- Define a program in a folder on the desktop
- Define changes to the MMPM2.INI file
- Define changes to other INI files

Defining a Program in a Folder on the Desktop

Use the WPObject structure to define a folder or a program (to be added to a folder). The WPObject structure calls the OS/2 WinCreateObject function, which adds an icon and title to the desktop. This structure indirectly changes the OS2.INI file. Refer to the *OS/2 Presentation Manager Guide and Reference* for object class definitions and supported keywords for the object class you are creating.

Use the WPObject structure shown in the following example to define a folder.

```
WPObject =
    (
    WPClassName   = "WPFolder"
```

```
    WPTitle       = "title"
    WPSetupString = "ICONFILE=$(DEST)icon;OBJECTID=<folderobjid>"
    WPLocation    = "<parentfolderobjid>"
    WPFlags       = wpflags
    )
```

title	Specifies the folder title to be displayed below the object.
icon	Specifies the file name of the icon representing the object.
<folderobjid>	Specifies the OS/2 unique object ID that is used to find this folder. This is used by the Workplace Shell to determine if this folder exists or not. It is also used in the *WPLocation* field of other *WPObject* definitions.
<parentfolderobjid>	Specifies the folder object ID of the folder in which this folder is to be placed. For example, "<WP_DESKTOP>" is the object ID for the Workplace Shell Desktop folder.
wpflags	This value specifies what action is to be taken if the object already exists. 0 - Fail if the object exists. 1 - Replace the object if it exists. 2 - Update the object if it exists (change the specified fields to the given values).

In the following example, a folder called Multimedia Presentation Manager Toolkit/2 will be added to the desktop.

```
WPObject =
    (
    WPClassName   = "WPFolder"
    WPTitle       = "Multimedia Presentation\nManager Toolkit/2"
    WPSetupString = "ICONFILE=$(DEST)MMTOOLKT.ICO;OBJECTID=<MMPMTLK>"
    WPLocation    = "<WP_DESKTOP>"
    WPFlags       = 2L
    )
```

You can also use the WPObject structure to define a program that will be added to a folder on the desktop as shown in the following figure.

```
WPObject =
(
WPClassName   = "WPProgram"
WPTitle       = "title"
WPSetupString = "EXENAME=path file;STARTUPDIR =dir;PROGTYPE=PM;
     ICONFILE=$(DEST)icon; [ASSOCTYPE =type;]
     [ASSOCFILTER=filter;] OBJECTID=<pgmobjid>"
WPLocation    = "<parentfolderobjid>"
WPFlags       = wpflags
)
```

title	Specifies the title to be displayed below the object in the parent folder.
path	Specifies a supported macro or the full path for the EXE file.
file	Specifies the EXE file name.
dir	Specifies the full path of the startup directory or the macro $(DIR)*#* (where *#* is a defined destination directory in CONTROL.SCR).
icon	Specifies the file name of the icon representing the object.
type	Specifies one or more association types such as "Waveform." Multiple values are separated by commas.
filter	Specifies one or more association filter types such as "*.WAV." Multiple values are separated by commas.
<pgmobjid>	Specifies the OS/2 unique object ID that is used to find this program. This is used by the installation program to determine if this program exists in the parent folder. It is not used

in the *WPLocation* field of any *WPObject* definition.

<parentfolderobjid> Specifies the folder object ID of the folder in which this program is to be placed.

wpflags This value specifies what action is to be taken if the object already exists.

- 0 - Fail if the object exists.
- 1 - Replace the object if it exists.
- 2 - Update the object if it exists (change the specified fields to the given values).

JoinEA Structure

The JoinEA structure shown in the following example causes the joining of an EA file to the parent file or directory. If the file or directory is used in a WPObject declaration, this structure should precede that declaration.

```
JoinEA =
    (
    JoinFileName   = "full path to file"
    JoinEAFileName = "full path to EA file"
    )
```

full path to file Specifies the full path to this file. Supported macros may be used.

full path to EA file Specifies the full path to the EA file. Supported macros may be used.

JoinLongNameEA Structure

The JoinLongNameEA structure shown in the following example allows you to specify a name that is greater than the standard 8-character length for a directory you are going to create as a Workplace Shell object. The JoinLongNameEA statement causes an EA file of type LONGNAME to be added to the directory. The long name is then displayed whenever the directory appears as a folder object. The JoinLongNameEA statement should come before creating the directory as a Workplace Shell object.

```
JoinLongNameEA =
    (
    JoinLongName       = "longname"
    JoinLongFileName   = "full path to directory"
    JoinEALongFileName = "full path to EA file"
    )
```

longname Specifies the new long name to be displayed.

full path to directory Specifies the directory to which EAs are to be attached. Supported macros may be used.

full path to EA file Specifies the full path to the EA file. Supported macros may be used.

Following is an example of a long name specified for an OS/2 multimedia directory with the JoinLongNameEA structure. The long name "Sound Bites" is added to the directory defined as "9" in the CONTROL.SCR file. In this case, it is the "Sounds" directory.

```
JoinLongNameEA =
    (
    JoinLongName       = "Sound Bites"
    JoinLongFileName   = "$(DIR)9"
    JoinEALongFileName = "$(DRIVE):\\MMOS2\\INSTALL\\sounds.eas"
    )
```

The $(DEST) macro is not used for this structure because the file is created, not copied and changed by MINSTALL. Notice that the file is placed in the \MMOS2\INSTALL subdirectory. This is the directory to which all EAs should be copied.

Defining Changes to the MMPM2.INI File

Some multimedia subsystems require unique information in the MMPM2.INI file. This file is used by the media device manager (MDM) to maintain a database of installed multimedia components and devices.

The following structures are used to make changes to the MMPM2.INI file.
MciInstallDrv Structure

The MciInstallDrv structure shown in the following example allows you to install MCDs on your system.

```
MciInstallDrv =
    (
    DrvInstallName   = "internalname"
    DrvDeviceType    =  devicetypecode
    DrvDeviceFlag    =  deviceflag
    DrvVersionNumber = "verno"
    DrvProductInfo   = "name2"
    DrvMCDDriver     = "mcdname"
    DrvVSDDriver     = "vsdname"
    DrvPDDName       = "pddname"
    DrvMCDTable      = "mcdtablename"
    DrvVSDTable      = "vsdtablename"
    DrvShareType     = number1
    DrvResourceName  = "resname"
    DrvResourceUnits = number2
    DrvClassArray[num] =
           (
                (DrvClassNumber = number3)
           )
    )
```

internalname	Specifies the name under which the device is being installed. This must be a unique name. Consider using a name that is a combination of a company name, device type, and device model. For example, the IBM media driver for the Sound Blaster waveaudio device is ibmwavesb01.
devicetypecode	Specifies an encoding of the device type. These codes are defined in the MCIOS2.H file located in the \TOOLKIT\H subdirectory.
deviceflag	Specifies the LONG INT with the flags set. This determines whether or not the device is controllable.
verno	Specifies the version number in the format "*dd.dd.dd*" where *dd* represents digits.
name2	Specifies the full name of the product and can be translated.
mcdname	Specifies the name of the DLL that contains the media control driver (MCD). This driver is loaded by the media device manager (MDM).
vsdname	Specifies the name of the DLL that contains the vendor-specific driver (VSD) used by the MCD (if any).
pddname	Specifies the name of the physical device driver used by the MCD (if any).
mcdtablename	Specifies the name of the DLL containing the MCD command table. (OS/2 multimedia provides a standard command table.)
vsdtablename	Specifies the name of the DLL containing the VSD command table.
number1	Specifies an encoding of the valid types of sharing supported. These codes are defined in the MMDRVOS2.H file located in the \TOOLKIT\H subdirectory.
resname	Specifies a unique name for the management of the driver resources.
number2	Specifies the maximum number of resource units supported for the driver.
num	Specifies the number of resource classes defined in the next field.
number3	Specifies the maximum number of resource units used by the class.

MciInstallAlias Structure

The MciInstallAlias structure shown in the following example allows you to specify an alternate name for a driver installed on your system.

```
MciInstallAlias =
    (
    AliasInstallName = "internalname"
    AliasString      = "aliasstring"
    )
```

internalname Specifies the internal name of the driver with which the alias is associated. This name is specified in the MciInstallDrv structure.

aliasstring Specifies an alternate name of the driver.

MciInstallConn Structure

Each implementation of a media driver defines certain paths of information flow into and out of the device. These paths are known as *connectors* . Connectors have defined connector types, and each connector type has an associated connector-type name. The MciInstallConn structure shown in the following example allows you to install the media connectors on your system.

```
MciInstallConn =
    (
    ConnInstallName = "internalname"
    ConnArray [num1]
         (
              (
              ConnType      = num2
              ConnInstallTo = "connto"
              ConnIndexTo   = num3
              )
         )
    )
```

internalname Specifies the internal name of the driver with which the connector is associated. This name is specified in the MciInstallDrv structure.

num1 Specifies the number of entries in the array.

num2 Specifies the connection type. Connection types are defined in the MCIOS2.H file located in the \TOOLKIT\H subdirectory.

connto Specifies the internal name of the driver to connect to.

num3 Specifies the connector index to the other driver specified in *connto* .

MciInstallExt Structure

When an element name is specified as the device name on an MCI_OPEN message and no device type is specified, the device type is identified by the file extension. For example, if the .WAV extension is associated with an internal driver name, that driver will be used if a file ending in .WAV is opened.

The MciInstallExt structure shown below allows you to define media control interface file extensions on your system.

```
MciInstallExt =
    (
    ExtInstallName = "internalname"
    ExtArray[num]  =
         (
         (ExtString = "string")
         )
    )
```

internalname Specifies the internal name of the driver with which the extension is associated. This name is specified in the MciInstallDrv structure.

num Specifies the number of external strings defined in the *string* field.

string Specifies the valid extensions.

MciInstallParm Structure

The MciInstallParm structure shown below allows you to define device-specific parameters that provide additional information about the driver to your MCD. For example, this structure is used to define to the MIDI MCD which MIDI map table to use for each sequencer.

```
MciInstallParm =
   (
   ParmInstallName = "internalname"
   ParmString      = "device specific parameters"
   )
```

internalname	Specifies the internal name of the driver with which the parameters are associated. This name is specified in the MciInstallDrv structure.
device specific parameters	Specifies the parameters needed by the driver. These parameters can be used to provide additional information about the driver.

Defining Changes to Other INI Files

You can create an INI change control file to make changes to any INI file that a driver needs. For example, you could initialize an OS/2 profile, define MIDI maps, and define external pages. You define changes to INI files using the ProfileData structure.

The following shows an example of the ProfileData structure.

```
   ProfileData =
      (
      ini="inifilename"
      appname="appname"
      keyname="keyname"
      dll="resourcedllname"
      id=resourceid
      )
```

inifilename	Specifies the name of an OS/2 INI file. For example, MIDITYPE.INI.
appname	Specifies the value of the appname parameter. This is the appname to be used in the INI file.
keyname	Specifies the unique name (the variable name or keyname) of the item being installed
resourcedllname	Specifies the name of the DLL that contains a RCDATA resource with the ID identified in *resourceid* .
resourceid	Specifies the resource ID of the RCDATA resource (where the value is stored in the RC file). The *resourceid* is a LONG numeric value. For example, 120L.

Writing an Installation DLL

You can provide one or two installation DLLs. These DLLs can be the same DLL with two different entry points or two different DLLs with corresponding entry points. There are two ways to call an installation DLL:

- Call the routine after all files have been copied, but *before* script files have been processed (using the SSDLL and SSDLLENTRY keywords), or
- Call the routine after all files have been copied and *after* script files have been processed (using the SSTERMDLL and SSTERMDLLENTRY keywords).

The parameters for each entry point are as follows:

HWND (input)	Owner handle. This handle allows the DLL to create windows for the user interface.
PSZ (input)	Source path of the installation package.
PSZ (input)	Target drive (a drive letter and colon (for example, *d*:).
PSZ (input)	DLL input parameters, as specified by the SSDLLINPUTPARMS keyword in CONTROL.SCR.
HWND (input)	MINSTALL object window handle that receives messages to perform media control interface and CONFIG.SYS operations.
PSZ (output)	A CHAR[256] null-terminated string that contains response-file data needed by the DLL. The encoded string is created by the DLL as a series of ASCII characters. This information allows MINSTALL to operate in an unattended installation mode where all user responses are provided by the response-file string. The encoded information is passed to the DLL, and all user interaction is bypassed.

The following is an example prototype used to define an installation DLL.

```
ULONG APIENTRY StartMyInstall (HWND hwndOwnerHandle,
                               PSZ pszSourcePath,
                               PSZ pszTargetDrive,
                               PSZ pszMyParms,
                               HWND hwndMinstallHandle,
                               PSZ pszResponseFile);
```

MINSTALL provides numerous services to the installation DLLs due to the various environments in which MINSTALL must operate. You can use MINSTALL on a machine that has never had OS/2 multimedia installed on it. You can also use MINSTALL on a machine that has OS/2 multimedia installed on the hard drive. OS/2 multimedia does not need to be running to use MINSTALL in this environment. MINSTALL retains control of the MMPM2.INI and CONFIG.SYS files to ensure no changes are made to these files while MINSTALL is running.

While MINSTALL has control of the MMPM2.INI and CONFIG.SYS files it might be necessary for an installation DLL to read or write to one or both of these files. MINSTALL provides an interface to allow the installation DLLs access to both the MMPM2.INI and CONFIG.SYS files. Also, if OS/2 multimedia is installed and running, some files might be open that MINSTALL attempts to replace. When this occurs, MINSTALL copies the open files to a temporary directory until the next system restart.

The following table lists the messages, along with the message formats and descriptions, available to an installation DLL. An installation DLL can send these messages with WinSendMsg or WinPostMsg.

Message	**Description**
IM_CODEC1INSTALL	Installs a CODEC using the *ulCodecCompType* field.

```
mp1 = 0;                                /* Not used */
mp2 = MPFROMP(PINSTCODECINIFILEINFO);  /* Pointer to the
                                          INSTIOPROC structure */
```

IM_CODEC2INSTALL	Installs a CODEC using the *fccCodecCompType[5]* field.

```
mp1 = 0;                                 /* Not used */
mp2 = MPFROMP (PINSTCODECINIFILEINFO);  /* Pointer to the
                                           INSTIOPROC structure */
```

IM_CONFIGDELETE	Deletes a line from the CONFIG.SYS file.

```
mp1 = MPFROMP(PCONFIGDATA); /* Pointer to the CONFIGDATA structure */
mp2 = 0;                    /* Not used */
```

IM_CONFIGENUMERATE	Gets a line from the CONFIG.SYS file.

```
mp1 = MPFROMP(PCONFIGDATA); /* Pointer to the CONFIGDATA structure */
```

```
mp2 = 0;                    /* Not used */
```

IM_CONFIGMERGE — Merges data into an existing CONFIG.SYS entry.

```
mp1 = MPFROMP(PCONFIGDATA); /* Pointer to the CONFIGDATA structure */
mp2 = 0;                    /* Not used */
```

IM_CONFIGNEW — Adds a new line to the CONFIG.SYS file.

```
mp1 = MPFROMP(PCONFIGDATA); /* Pointer to the CONFIGDATA structure */
mp2 = 0;                    /* Not used */
```

IM_CONFIGQUERYCHANGED — Returns TRUE if the CONFIG.SYS file has been changed.

```
mp1 = 0;  /* Not used */
mp2 = 0;  /* Not used */
```

IM_CONFIGREPLACE — Replaces an existing CONFIG.SYS file.

```
mp1 = MPFROMP(PCONFIGDATA); /* Pointer to the CONFIGDATA structure */
mp2 = 0;                    /* Not used */
```

IM_CONFIGUPDATE — Updates an existing CONFIG.SYS entry.

```
mp1 = MPFROMP(PCONFIGDATA); /* Pointer to the CONFIGDATA structure */
mp2 = 0;                    /* Not used */
```

IM_CREATE_WPS_OBJECT — Installs a folder and its contents.

```
mp1 = 0;                         /* Not used */
mp2 = MPFROMP(PINSTOBJECTDATA); /* Pointer to the
                                   INSTOBJECTDATA structure */
```

IM_DESTROY_WPS_OBJECT — Destroys an existing folder and its contents.

```
mp1 = 0;                /* Not used */
mp2 = MPFROMP(HOBJECT); /* This must be the exact OBJECTID
                           with which the object was created  */
```

IM_EA_JOIN — Joins an EA file to its parent file (the file that it was previously separated from).

```
mp1 = 0;                    /* Not used */
mp2 = MPFROMP(PINSTEAJOIN); /* Pointer to the INSTEAJOIN
                               structure */
```

IM_EA_LONG_NAME_JOIN — Creates an EA file containing a long name (greater than 8 characters but less than 256) and joins it to a file or directory.

```
mp1 = 0;                         /* Not used */
```

```
mp2 = MPFROMP(PINSTEALONGNAMEJOIN); /* A pointer to the structure
                                       that contains the long name
                                       as an ASCII string, the file or
                                       directory name to which the long
                                       name is to be applied, and the
                                       new EA name */
```

IM_LOG_ERROR — Writes a message to the MINSTALL.LOG file. This can be an error type message or only an information type message.

```
mp1 = MPFROMP((PSZ)pszStatement); /* The text of the message to insert
                                     at the end of the MINSTALL.LOG
                                     file */
mp2 = 0;                          /* Not used */
```

IM_MCI_EXTENDED_SYSINFO — Maps to the mciSendCommand MCI_SYSINFO.

```
mp1 = MPFROML(LONG);  /* The MCI_SYSINFO extended function desired */
mp2 = MPFROMP(*MCI_SYSINFO_PARMS); /* The SYSINFO
                                                    structure       */
```

IM_MCI_SEND_COMMAND — Sends an MCI command.

```
mp1 = 0;                             /* Not used */
mp2 = MPFROMP(PINSTMCISENDCOMMAND); /* Pointer to the INSTMCISENDCOMMAND
                                       structure */
```

IM_MIDIMAP_INSTALL — Installs a MIDI map.

```
mp1 = 0;                             /* Not used */
mp2 = MPFROMP(PMIDIMAPINSTALLDATA); /* Pointer to the
                                       MIDIMAPINSTALLDATA structure */
```

IM_MMIO_INSTALL — Installs an IOProc.

```
mp1 = 0;                    /* Not used */
mp2 = MPFROMP(PINSTIOPROC); /* Pointer to the INSTIOPROC structure */
```

IM_QUERYPATH — Requests the current location of a copied file.

```
mp1 = PMFROMP(PSZ); /* The name of the file needed  */
mp2 = PMFROMP(PSZ); /* The full path to the file    */
```

IM_SPI_INSTALL — Installs stream protocol information.

```
mp1 = 0;            /* Not used */
mp2 = MPFROMP(PSZ); /* The fully qualified path of a SPI resource DLL */
```

The data structures used as parameters for the messages in the previous list are defined in the MINSTALL.H file located in the \TOOLKIT\H subdirectory.

Follow these guidelines when writing an installation DLL:

- Set the mouse pointer to SPTR_WAIT during any operation lasting more than one second between requests for user information.
- Keep the user interface as responsive as possible by using the MM_DISPATCHVARS and MM_DISPATCHMESSAGES() macros before and after all I/O intensive operations and and while updates are being made to INI files and the CONFIG.SYS file. This is necessary because installation DLLs are executed in the MINSTALL message queue thread. These macros are defined in the MINSTALL.H file.

Installing a Media Control Driver

To install a media control driver (MCD) in the OS/2 multimedia system:

1. Create an INI change control file containing the information needed for the media control driver. This will make the necessary changes in the MMPM2.INI file.

   When you create a new MCD, you have to install drivers into the OS/2 multimedia system to use the new MCD. The following is an example of how to use structures in an INI Change Control File to install a Sound Blaster Waveform Audio Driver that uses a new Audio MCD named "MyNewMCD." See Defining Changes to the MMPM2.INI File for a detailed description of these structures.

```
MciInstallDrv =
    (
    DrvInstallName     = "MyWaveSB01"
    DrvDeviceType      = 7
    DrvDeviceFlag      = 01L
    DrvVersionNumber   = "1"
    DrvProductInfo     = "Sound Blaster Pro MCV"
    DrvMCDDriver       = "MyNewMCD"
    DrvVSDDriver       = "Audioif"
    DrvPDDName         = "SBAUD1$"
    DrvMCDTable        = "MDM"
    DrvVSDTable        = ""
    DrvShareType       = 3
    DrvResourceName    = "SoundblasterW01"
    DrvResourceUnits   = 1
    DrvClassArray[1]  =
         (
              ( DrvClassNumber = 1 )
         )
    )
MciInstallParm =
    (
    ParmInstallName   = "MyWaveSB01"
    ParmString        = "FORMAT=1,SAMPRATE=22050,BPS=8,CHANNELS=2,
                        DIRECTION=PLAY"
    )


MciInstallConn =
    (
    ConnInstallName="MyWaveSB01"
    ConnArray[1]=
         (
            (
            ConnType=3                  /* Wavestream connector */
            ConnInstallTo="MyAmpMixSB01" /* Connect to ampmixer  */
            ConnIndexTo=1               /* First connector in
                                           ampmixer             */
            )
         )
    )


MciInstallAlias =
    (
    AliasInstallName="MyWaveSB01"
    AliasString="Wave Audio"
    )
```

```
MciInstallExt =
    (
    ExtInstallName = "MyWaveSB01"
    ExtArray[1] =
       (
          (ExtString = "WAV")
       )
    )
```

2. Install external settings pages in the Multimedia Setup notebook (optional). See Inserting External Settings Pages.

3. Specify the name of your INI change control file in an SSINICH keyword in the master control file (CONTROL.SCR). See CONTROL.SCR Subsystem Definition for more information. An example of a CONTROL.SCR file and a description of the keywords are provided in that section. For example:

```
SSINICH="MYMCD.SCR"
```

You might want to experiment with the sample files provided in the \TOOLKIT\SAMPLES\MM\CF subdirectory. If you decide to experiment with the MCD templates provided in the toolkit, you can test your changes by replacing the name of the driver that comes with OS/2 multimedia with the name of your driver. For example, you can edit the MMPM2.INI file and change the `MCDDRIVER=AUDIOMCD` statement to `MCDDRIVER=AUDIOMCT`. You also must copy your MCD to a directory that is included in the LIBPATH statement of your CONFIG.SYS file (for example, \MMOS2\DLL). After you make these changes and restart your system, OS/2 multimedia will use your MCD instead of the OS/2 multimedia MCD.

Note: After you complete your MCD testing, you must change the MMPM2.INI text file back to its original state. Unpredictable results can occur when OS/2 multimedia is used without its supported MCDs.

Installing a Stream Handler

When a stream handler is installed by MINSTALL, the following events occur:

1. A media driver issues an SpiGetHandler function.
2. When the handler is loaded, the Sync/Stream Manager (SSM) reads in the respective stream protocol control blocks (SPCBs) from the SPI.INI file.
3. If the stream handler is a DLL, the SSM loads and registers the DLL with the DOSLoadModule function. However, if the stream handler is a device driver, the SSM does not issue a DOSLoadModule because a CONFIG.SYS statement installs the device drivers during system startup.

Before the SSM can read in the respective SPCBs, the SPCBs first must be installed in the SPI.INI file, which is built during the multimedia installation of the base OS/2 product. This file contains all the information about stream handlers and stream classes that are currently supported by OS/2 multimedia.

Creating a Resource File

To install SPCB information in the SPI.INI file, you first must create a resource file containing all the information about the stream handlers to update or add to the SPI.INI profile. The fields and the order of these fields should match the example shown in the example that follows.

```
#include <os2.h>
#include <os2me.h>

#define  SPI_RESOURCE1                    (SPI_RESOURCE + 1)
#define  SPI_RESOURCE2                    (SPI_RESOURCE + 2)

#define  SPCBHAND_RCVSYNC_GENSYNC_GENTIME
                    (SPCBHAND_RCVSYNC + SPCBHAND_GENSYNC + SPCBHAND_GENTIME)
```

```
RCDATA   SPI_RESOURCE
BEGIN
  2                     /* number of stream handlers resources */
END

RCDATA   SPI_RESOURCE1
  BEGIN
     "TESTSYS\0",                       /* Class name      */
     "R3TEST\0",                        /* Handler name    */
      SH_DLL_TYPE,                      /* PDD or DLL flag */
      "R3TEST\0",                       /* PDD or DLL name */
       1,                               /* Number of SPCBs */
       SPCBSIZE,                        /* Length of SPCB  */
       DATATYPE_GENERIC,                /* Data type       */
       SUBTYPE_NONE,                    /* Sub type        */
       0L,                             /* Internal key     */
       0L,                             /* Data flag        */
       0L,                             /* # of records     */
       1L,                             /* Block size       */
       4096L,                          /* Buffer size      */
       2L,                             /* Min # of buffers */
       4L,                             /* Max # of buffers */
       1L,                             /* # empty buffs to start src */
       2L,                             /* # full buffs to start tgt */
       SPCBBUF_NONCONTIGUOUS,          /* Buffer flag  */
       SPCBHAND_RCVSYNC,               /* Handler flag */
       0L,                             /* Sync tolerance value */
       0L,                             /* Save sync pulse generation */
       0L,                             /* Bytes/unit of time       */
           0L                          /* MMTIME each unit represents */
  END
RCDATA   SPI_RESOURCE2
  BEGIN
     "TESTSYS\0",               /* Class name      */
     "R0TEST\0",                /* Handler name    */
      SH_PDD_TYPE,              /* PDD or DLL flag */
     "R0TEST.SYS\0",            /* PDD or DLL name */
      2,                        /* Number of SPCBs */
      SPCBSIZE,                 /* Length of SPCB  */
      DATATYPE_ADPCM_AVC,       /* Data type       */
      0L,                       /* Sub type     */
      0L,                       /* Internal key */
      SPCBDATA_CUETIME,         /* Data flag    */
      0L,                       /* # of records */
      1L,                       /* Block size   */
      4096L,                    /* Buffer size  */
      10L,                      /* Min # of buffers */
      10L,                      /* Max # of buffers */
      1L,                       /* # empty buffs to start src */
      1L,                       /* # full buffs to start tgt  */
      SPCBBUF_NONCONTIGUOUS,           /* Buffer flag  */
      SPCBHAND_RCVSYNC_GENSYNC_GENTIME, /* Handler flag */
      0L,                       /* Sync tolerance value */
      0L,                       /* Save sync pulse generation */
      0L,                       /* Bytes/unit of time */
      0L,                       /* MMTIME each unit represents */

      SPCBSIZE,                 /* Length of SPCB */
      DATATYPE_WAVEFORM,        /* Data type    */
      0L,                       /* Sub type     */
      0L,                       /* Internal key */
      SPCBDATA_CUETIME,         /* Data flag    */
      0L,                       /* # of records */
      1L,                       /* Block size   */
      8192L,                    /* Buffer size  */
      10L,                      /* Min # of buffers */
      10L,                      /* Max # of buffers */
      1L,                       /* # empty buffs to start src */
      1L,                       /* # full buffs to start tgt  */
      SPCBBUF_NONCONTIGUOUS,    /* Buffer flag */
      SPCBHAND_RCVSYNC_GENSYNC_GENTIME, /* Handler flag */
      0L,                       /* Sync tolerance value   */
      0L,                       /* Save sync pulse generation */
      0L,                       /* Bytes/unit of time */
      0L                        /* MMTIME each unit represents */
  END
```

Creating a Stub Routine

After you create a resource file, you must create a stub routine to generate the DLL that contains the resource file. The following example shows an example of a sample stub routine (RCSTUB.C) that is used to create the DLL containing the resource shown in the previous example.

```
#include <os2.h>
VOID RCSTUB()
{
}
```

Building the DLL Containing the Resource

Next, you must build the DLL containing the resource. A DLL is generated that will be used in Updating the SPI.INI File to update the SPI.INI profile. The DLL is built by entering the following command:

```
NMAKE /F MAKERES.MAK
```

The following example shows a sample makefile (MAKERES.MAK), which is used to build the TESTRES.DLL containing the resource shown in the previous example. (See Creating a Resource File.)

```
.SUFFIXES: .com .sys .exe .obj .mbj .asm .inc .def .lrf .crf .ref \
           .lst .sym .map .c .h .lib .msg .pro .txt .cod .cvk
RCDLL=testres
/*****************************/
/* Compiler and Tools location */
/*****************************/
MSRC     =..
TOOLS    =..\..\TOOLS
SHIP_LIB =..\..\SHIP_LIB
SHIP_H   =..\SHIP_H
SHIP_INC =..\SHIP_INC
COMMON   =..\COMMON
INC      =..\..\SRC\INC
H        =..\..\SRC\H
H386     =..\..\SRC\H386
LIB      =..\..\SRC\LIB
LIB386   =..\..\SRC\LIB386

/*****************************/
/* Definitions for C Compiler */
/*****************************/
CCOMP386=cl386
CFLAGS386= /c /G3 /AS /W3 /Od /DLINT_ARGS
CINC386=-I. -I$(SHIP_H) -I$(COMMON) -I$(H386) -I$(H386)\SYS -I$(H) -I$(H)\SYS

/*****************************/
/* Definitions for linker     */
/*****************************/
LINK386  =link386
LFLAGS386= $(LNK_DEBUG) /batch /map /nod /noi /packcode /packdata
LIBS386  =$(NAMELIB) os2386 libc doscalls

/***********************************/
/* Definitions for RESOURCE compiler */
/***********************************/
RC       = rc
RCINC    = -i $(H) -i $(SHIP_H) -i $(COMMON)

/*************************/
/*  Object file lists     */
/*************************/

RCOBJS   =$(COMMON)\rcstub.obj

/***********************************/
```

```
/* Inference Rules                    */
/*************************************/
.c.obj:
        $(CCOMP386) $(CFLAGS386) $(CINC386) /Fo$(<R).obj $(C_LST) $(<R).c

/*************************************/
/* Target Descriptions               */
/*************************************/
!include        "$(H)\common.mak"

all: rc

/*************************************/
/* SSMRES.DLL Target Descriptions    */
/*************************************/
rc: $(RCDLL).dll

$(RCDLL).dll:  $(RCOBJS) $(RCDLL).rc makeres.mak $(RCDLL).lrf  \
               $(RCDLL).def
        $(LINK386) $(LFLAGS386) @$(RCDLL).lrf
        $(RC) $(RCINC) $(RCDLL).rc $(RCDLL).dll
#
# Make DEF file
#
$(RCDLL).def:  makeres.mak
        @echo Creating file <<$(@B).def
LIBRARY $(RCDLL)
DESCRIPTION 'DLL file containing resources'
STUB 'OS2STUB.EXE'
DATA NONE
<<keep

#
# Make link response file
#
$(RCDLL).lrf: makeres.mak
        @echo Creating file <<$(@B).lrf
$(RCOBJS)
$(RCDLL).dll
$(RCDLL).map $(LFLAGS386)
os2386 libcdll
$(RCDLL).def;
<<keep
```

Updating the SPI.INI File

The resource DLL that was built by the makefile must now be used with the master control file, the file list control file, and the SpiInstall structure to update the SPI.INI profile. (See Master Control File and File List Control File for information about the master control file and the file list control file.) Following are examples of the files needed to update the SPI.INI file:

- INI change control file
- CONTROL.SCR entry
- FILELIST.TLK entry

These files must be placed on a diskette with the TESTRES.DLL file that was created.

The following example shows how to write an INI change control file named TEST.SCR to install the DLL file (TESTRES.DLL).

```
SpiInstall =
    (
    SpiDllName = "$(DEST)TESTRES.DLL"
    )
```

The following example shows how to specify the TEST.SCR INI change control file in the CONTROL.SCR file.

```
package   ="SPI.INI Update"
```

```
codepage  =437
filelist  ="TEST.MMI"
groupcount=2
munitcount=1
medianame ="SPI.INI Update Disk 1"
sourcedir = "\\"                    = 0
destindir = "\\MMOS2\\DLL\\"        = 2
destindir = "\\MMOS2\\INSTALL\\"    = 4

ssgroup   =0
ssname    ="Base"
ssversion ="1.0.0"
sssize    =10

ssgroup   =1
ssname    ="SPI.INI Update"
ssversion ="1.0.0"
sssize    =10
ssinich   ="TEST.SCR"
```

The following is an example of how to specify the TEST.SCR INI change control file in the FILELIST.TLK file.

```
/* Total number of entries is 3                    */

     3

/* Disk# Group# Dest#    Path             FileName  */

    0      1     2       0               "TESTRES.DLL"
    0      1     4       0               "TEST.SCR"
    0      0     4       0               "TEST.SCR"
```

Installing Stream Protocol

A stream handler must also support SpiInstallProtocol to be able to receive the SPCB's. For example, suppose an application or media driver wants to install another SPCB of the same data type and subtype as an SPCB that it already has installed. The application or media driver will need to assign a new internal key value, that is used to differentiate between multiple SPCB's of the same data stream type. All the default SPCBs are installed with an internal key value of 0. Any additional SPCBs that overlap any of the data types that are already installed, need to use a different internal key value. Therefore, the stream handler must have code that allows for this scenario.

The sample code in the following example illustrates how to install stream protocol.

```
RC ShcInstallProtocol(pipparm)
PPARM_INSTPROT pipparm;

{ /* Start of ShcInstallProtocol */

RC rc = NO_ERROR;                       /* local return code */
int notfound = TRUE;
PESPCB pTempEspcb;
PESPCB pPrevEspcb;

 /* the ESPCB list is under semaphore control */

 if (!(rc = DosRequestMutexSem(hmtxGlobalData, SEM_INDEFINITE_WAIT)))
   { /* obtained semaphore */
     if (pipparm->ulFlag & SPI_DEINSTALL_PROTOCOL)
       { /* DeInstall */

         /* To Deinstall, Find the spcb,                 */
         /*               Take it off the espcb chain,   */
         /*               Free the espcb memory allocated */

         rc = ERROR_INVALID_SPCBKEY;
         pPrevEspcb = NULL;
         pTempEspcb = pESPCB_ListHead;
```

```
        while (pTempEspcb && notfound)
          { /* Loop thru espcbs */
            if ((pipparm->spcbkey.ulDataType ==
                        pTempEspcb->spcb.spcbkey.ulDataType) &&
                (pipparm->spcbkey.ulDataSubType ==
                        pTempEspcb->spcb.spcbkey.ulDataSubType) &&
                (pipparm->spcbkey.ulIntKey ==
                        pTempEspcb->spcb.spcbkey.ulIntKey))
              { /* found match */
                notfound = FALSE;
                rc = NO_ERROR;

                /* Take the espcb off the chain */

                if (pPrevEspcb)
                  {
                    pPrevEspcb->pnxtESPCB = pTempEspcb->pnxtESPCB;
                  }
                else
                  {
                    pESPCB_ListHead = pTempEspcb->pnxtESPCB;
                  }
                HhpFreeMem(hHeap, pTempEspcb);

              } /* found match */
            else
              { /* try the next espcb in the chain */

                pPrevEspcb = pTempEspcb;
                pTempEspcb = pTempEspcb->pnxtESPCB;

              } /* try the next espcb in the chain */
          } /* Loop thru espcbs */
      } /* DeInstall */
    else
      { /* Install */

        /* If the SPCB already exists then error */

        if (ShFindEspcb(pipparm->spcbkey))
          {
            rc = ERROR_INVALID_SPCBKEY;
          }
        else
          { /* OK to add spcb */

            /* Allocate the espcb and put it on the front of the chain */

            pTempEspcb = (PESPCB)HhpAllocMem(hHeap, sizeof(ESPCB));
            if (pTempEspcb)
              {
                pTempEspcb->spcb = *(pipparm->pspcb);
                pTempEspcb->pnxtESPCB = pESPCB_ListHead;
                pESPCB_ListHead = pTempEspcb;
              }
            else
              }
                rc = ERROR_ALLOC_RESOURCES;
              }
          } /* OK to add spcb */
      } /* Install */

    DosReleaseMutexSem(hmtxGlobalData);

  } /* obtained semaphore */

 return(rc);

} /* End of ShcInstallProtocol */
```

Note: For descriptions of the parameters in the SPCB, see Stream Handlers.

Installing an I/O Procedure

To install an I/O procedure, an IOProc entry is added to the MMPMMMIO.INI file. This is accomplished by either writing an INI change control file or writing a program using the mmioIniFileHandler function. The IOProc is installed in the IOProc table ahead of the system-provided storage system IOProcs (DOS, MEM, and CF).

The mmioInstall structure shown in the following example allows you to install an IOProc in the system. The MMPMMMIO.INI file is modified with the latest mmioInstall data when the MINSTALL program is executed.

```
mmioInstall =
    (
    mmioFourCC        = "fourcc"
    mmioDllName       = "full path"
    mmioDllEntryPoint = "entry name"
    mmioFlags         = "flags"
    mmioExtendLen     = "extend length"
    mmioMediaType     = "media type"
    mmioIOProcType    = "file format"
    mmioDefExt        = "default extension"
    )
```

fourcc	Specifies the FOURCC of the I/O procedure.
full path	Specifies the full path to the file and the file name of the IOProc DLL. (You can use supported macros described in Supported Macros.)
entry name	Specifies the entry point of the IOProc DLL.
flags	Specifies any additional MMIO flags. Set to 0L for this release.
extend length	Specifies the extended length of 16 for this release.
media type	Specifies the media type of this file.
file format	Specifies the type of IOProc, either a file format or storage system IOProc.
default extension	Specifies the default file extension.

For example, to install the AVC audio I/O procedure, create an INI change control specifying the mmioInstall structure.

```
mmioInstall =
   (
   mmioFourCC        = "AVCA"
   mmioDllName       = "$(DEST)AVCAPROC.DLL"
   mmioDllEntryPoint = "AVCAIOProc"
   mmioFlags         = 0L
   mmioExtendLen     = 16L
   mmioMediaType     = 2L
   mmioIOProcType    = 2L
   mmioDefExt        = ""
   )
```

Specify the name of your INI change control file using the SSINICH keyword in the master control file (CONTROL.SCR). See CONTROL.SCR Subsystem Definition for an example of a CONTROL.SCR file and a description of the keywords. For example:

```
SSINICH="BASE1.SCR"
```

BASE1.SCR, located in the \MMOS2\INSTALL subdirectory, contains mmioInstall structure examples. You can also install an IOProc in your system by identifying the IOProc in the initialization file (MMPMMMIO.INI) using the mmioIniFileHandler function.

The following shows an example of how an application uses the mmioIniFileHandler function to install the OS/2 1.3 PM bitmap image IOProc.

```
#define FOURCC_OS13   mmioFOURCC( 'O', 'S', '1', '3' )

#pragma linkage( mmioIniFileHandler, system )
```

```
void main ()
{
 ULONG   rc;
 MMINIFILEINFO  mminifileinfo;
 mminifileinfo.fccIOProc = FOURCC_OS13;
 strcpy (mminifileinfo.szDLLName, "OS13PROC");
 strcpy (mminifileinfo.szProcName, "OS13BITMAPIOPROC");
 mminifileinfo.ulExtendLen = 16L;
 mminifileinfo.ulFlags = 0L;
 mminifileinfo.ulMediaType = MMIO_MEDIA_IMAGE;
 mminifileinfo.ulIOProcType = MMIO_IOPROC_FILEFORMAT;
 strcpy (mminifileinfo.szDefExt, "");

 printf ("Installing OS/2 PM Bitmap (V1.3) IOProc\n");

 rc = mmioIniFileHandler (&mminifileinfo, MMIO_INSTALLPROC);
 switch (rc)
 {
   case MMIO_SUCCESS:
    printf ("Installing Complete\n");
    break;
   case MMIOERR_INVALID_PARAMETER:
    printf ("Error in this install program\n");
    break;
   case MMIOERR_INTERNAL_SYSTEM:
    printf ("OS/2 MPM System Error\n");
    break;
   case MMIOERR_NO_CORE:
    printf ("Memory unavailable for this IOProc\n");
    break;
   case MMIOERR_INI_OPEN:
    printf ("Unable to access the OS/2 MMPMMMIO.INI file\n");
    break;
   case MMIOERR_INVALID_FILENAME:
    printf ("Cannot find the file : OS13PROC.DLL\n");
    break;
   default:
    printf ("Unknown error attempting to install OS/2 Bitmap V(1.3)\n");
    break;
 }
}
```

The advantage of installing I/O procedures in the MMPMMMIO.INI file is to achieve application transparency; I/O procedures become built-in as soon as you restart your OS/2 multimedia application. Note that the IOProc must reside in a DLL file, although more than one IOProc can reside in the DLL if necessary.

Inserting External Settings Pages

The following approach shows you how to insert settings pages in the Multimedia Setup notebook, where the code for the settings pages exists outside the Multimedia Setup program itself. This is applicable in situations where you want to insert an external settings page for configuration items that are not strictly related to a particular MCD. For example, pages that apply to the system or to all media control interface devices of a particular class.

Note: To insert a settings page for a particular MCD, see Inserting Pages in the Multimedia Setup Notebook.

MDM uses the MMPM2.INI file to maintain a data base of installed multimedia components and devices. MMPM2.INI is initialized by the MINSTALL program. Additional sections in the MMPM2.INI file are initialized based on information contained in the keywords in an installation control file, or as requested by an application calling MDM to make the change.

You can write an INI change control file containing the ProfileData structure to define external pages by following these steps:

1. Write a function to create the settings page. The function should be of the prototype shown in the following example.

```
HWND InsertExamplePage(PMCI_DEVICESETTINGS_PARMS pMCIDevSettings)

{
  HWND  hwndPage;               /* Page window handle    */
  CHAR  szTabText[CCHMAXPATH];  /* Buffer for tab string */
```

```
    ULONG ulPageId;              /* Page Identifier       */

/*********************************************************************/
/*   Load a modeless secondary window                                */
/*********************************************************************/
    hwndPage    = WinLoadSecondaryWindow(
                          pMCIDevSettings->hwndNotebook,
                          pMCIDevSettings->hwndNotebook,
                          ExamplePageDlgProc,
                          vhmodMRI,
                          ID_EXAMPLE,
                          (PVOID)pMCIDevSettings);

    if (!hwndPage) return (NULL);

    ulPageId = (ULONG)WinSendMsg( pMCIDevSettings->hwndNotebook,
                          BKM_INSERTPAGE,
                          (ULONG)NULL,
                          MPFROM2SHORT( BKA_AUTOPAGESIZE |
                          BKA_MINOR, BKA_LAST ) );
/*********************************************************************/
/*  Associate a secondary window with a notebook page.               */
/*********************************************************************/

    WinSendMsg(pMCIDevSettings->hwndNotebook, BKM_SETPAGEWINDOWHWND,
              MPFROMP( ulPageId ), MPFROMLONG( hwndPage ) );

/*********************************************************************/
/*  Get Tab Text from DLL                                            */
/*********************************************************************/

    WinLoadString(WinQueryAnchorBlock( HWND_DESKTOP ), vhmodMRI,
           (USHORT)IDB_EXAMPLE, CCHMAXPATH, szTabText );

/*********************************************************************/
/*  Set Tab Text                                                     */
/*********************************************************************/

    WinSendMsg( pMCIDevSettings->hwndNotebook, BKM_SETTABTEXT,
          MPFROMP( ulPageId ), szTabText );

    return( hwndPage );

 }

 typedef struct {
    HWND hwndHelpInstance;
 } MMPAGEINFO;
 typedef MMPAGEINFO * PMMPAGEINFO;

/*********************************************************************/
/*  Modeless secondary window procedure                              */
/*********************************************************************/

 MRESULT EXPENTRY ExamplePageDlgProc (HWND hwnd, USHORT msg,
                                      MPARAM mp1, MPARAM mp2)

 PMMPAGEINFO pMMPageInfo = (PMMPAGEINFO)
                           WinQueryWindowPtr (hwnd, QWL_USER);

   switch (msg) {
      case WM_INITDLG:
         /* Initialize The Page */
         pMMPageInfo = (PMMPAGEINFO) malloc(sizeof(MMPAGEINFO));
         WinSetWindowPtr (hwnd, QWL_USER, pMMPageInfo);

         /* Create a Help Instance */
         pMMPageInfo->hwndHelpInstance = WinCreateHelpInstance(...);
         break;

      case WM_DESTROY:
         /* Clean up page window resources */
          WinDestroyHelpInstance (pMMPageInfo->hwndHelpInstance);
          free (pMMPageInfo);
          break;

      case WM_COMMAND:
          /* Process All Commands */
          return ((MRESULT) FALSE);
```

```
        break;

    case MM_TABHELP:
        /* Display help for a tab */
        if (pMMPageInfo->hwndHelpInstance) {
            WinSendMsg(
                pMMPageInfo->hwndHelpInstance,
                HM_DISPLAY_HELP,
                MPFROMSHORT(WinQueryWindowUShort(hwnd, QWS_ID)),
                HM_RESOURCEID );
         }
         break;

    case WM_CLOSE:
        return ((MRESULT) FALSE);
        break;

    case WM_HELP:
        if (pMMPageInfo->hwndHelpInstance) {
            WinSendMsg(
                 pMMPageInfo->hwndHelpInstance,
                 HM_DISPLAY_HELP,
                 (MPARAM) mp1,
                 HM_RESOURCEID );
         }
         return ((MRESULT)TRUE);
         break;

    case WM_TRANSLATEACCEL:
          return (WinDefWindowProc (hwnd, msg, mp1, mp2));
          break;

    case HM_QUERY_KEYS_HELP:
          return((MRESULT) IDH_HELPFORKEYS);
          break;

    }

    return (WinDefSecondaryWindowProc(hwnd, msg, mp1, mp2));
}
```

2. Create an INI change control file containing the ProfileData structure as shown in the following example.

```
ProfileData =
   (
   ini="$(DIR)0\\MMPM.INI"            /* Name of the INI file    */
   appname="STPM_SettingsPage:7"      /* External settings pages */
   keyname="UniqueName"               /* Name of the page        */
   dll="RESDLLNAME"                   /* Resource DLL name       */
   id=33L                             /* Resource ID             */
   )
```

The external settings pages are stored with the *appname* of `STPM_SettingsPage:#`, where # is the string equivalent of MCI_DEVICETYPE_*xxx* values. The # value of 0 indicates that the settings page is a system-oriented settings page. The *keyname* value for an external settings page can be any unique name that identifies the page. The *dll* value specifies the name of the DLL that contains a RCDATA resource with the value specified in the *id* field. The *id* field is the resource ID of the RCDATA resource. The value is a LONG numeric value; for example, 33L.

In addition to the INI change control file, you must load a resource DLL during installation. The value of the data item is a string that contains the module name and the entry point of the DLL that inserts a settings page. The value string in the example shown in the following example is "MakePage,InsertExamplePage".

```
RCDATA  33
BEGIN
  "MakePage,InsertExamplePage"
END
```

This script format loads a block of data from a RCDATA resource stored in a DLL, and writes it to a standard OS/2 INI file using the OS/2 PrfWriteProfileData function. The ProfileData structure is discussed in more detail in Defining Changes to Other INI Files.

Note: Entry points that are registered in this way will apply to all devices of the declared media control interface device type.

Therefore, if a settings page is intended to apply to a particular device, then the function should verify that it is dealing with the appropriate device.

3. Specify the name of the INI change control file created in step 2 in CONTROL.SCR.

4. If you are writing an installation DLL, an external settings page can be installed by calling the OS/2 PrfWriteProfileData function directly as shown in the following example.

```
hini = PrfOpenProfile("c:\mmos2\mmpm.ini");
PrfWriteProfileData (hini,
                     "STPM_SettingsPage:7",         /* Appname      */
                     "UniqueName",                  /* Keyname      */
                     "MakePage, InsertExamplePage", /* Value        */
                     "strlen("MakePage, InsertExamplePage")+1);
                                                    /* Value length */
PrfCloseProfile(hini);
```

Refer to the *OS/2 Presentation Manager Guide and Reference* for more information on the PrfWriteProfileData function.

Installation LOG File

During installation, MINSTALL writes an installation log file called MINSTALL.LOG. The LOG file records installation information and can be used to identify problems that occur during the installation procedure.

When you start the installation procedure, the LOG file defaults to the root directory of the startup (boot) disk. When the user selects the **Install** push button in the installation main selection window, the LOG file is moved to the destination directory. The LOG file can be printed using the PRINT command or viewed using the TYPE command. To view the LOG file on the screen, at an OS/2 command prompt, type:

```
TYPE d:\MMOS2\INSTALL\MINSTALL.LOG
```

where *d* is the destination drive selected for the installation.

If a problem occurs during installation, MINSTALL might display a screen with information about the problem. If you cannot solve the problem using the information on the screen, you can view the LOG file to determine how to handle the problem. The problem will not necessarily be the last entry in the LOG file.

If you determine that you need to call IBM for support, print the LOG file before you call. The IBM support personnel will need the LOG information to resolve your installation problem.

If the installation is successful, a Congratulations screen appears.

Note: You might receive a message that the installation was successful, but when you view the LOG file you notice that the file reported some errors. This means that the errors were resolved during the installation procedure and the errors reported in the LOG file can be ignored.

The following example shows a sample of a MINSTALL.LOG file for a successful installation.

```
Begin parsing master file - Z:\RT\CONTROL.SCR.
Z:\RT\CONTROL.SCR parsed successfully.
Begin parsing file list - Z:\RT\MASTERH3.RT
Z:\RT\MASTERH3.RT parsed successfully.
The following file was copied successfully: D:\MMOS2\README
The following file was copied successfully: D:\MMOS2\MINSTALL.EXE
.
.
.
The following file was copied successfully: D:\MMOS2\INSTALL\WAVEFILE.EAS
Loaded DLL D:\MMOS2\DLL\MMSND.DLL and calling entry point InstallMMSound.
Returned from DLL D:\MMOS2\DLL\MMSND.DLL.
Loaded DLL D:\MMOS2\DLL\QRYUM.DLL and calling entry point LocateUM.
An extended Sysinfo failed call with dwItem=1024.
An extended Sysinfo failed call with dwItem=4096.
Completed updating MMPM2.INI with Digital Video Player devices.
Returned from DLL D:\MMOS2\DLL\QRYUM.DLL.
```

```
Loaded DLL D:\MMOS2\DLL\QRYCD.DLL and calling entry point LocateCD.
An extended Sysinfo failed call with dwItem=4.
.
.
.
An extended Sysinfo failed call with dwItem=4.
Returned from DLL D:\MMOS2\DLL\QRYCD.DLL.
Loaded DLL D:\MMOS2\DLL\QRYAD.DLL and calling entry point LocateMAUDIO.
Completed updating CONFIG.SYS with M-Audio devices
An extended Sysinfo failed call with dwItem=1024.
An extended Sysinfo failed call with dwItem=1024.
Completed updating MMPM2.INI with M-Audio devices
Returned from DLL D:\MMOS2\DLL\QRYAD.DLL.
Starting to parse the file 'D:\MMOS2\INSTALL\BASE.SCR'.
Successfully parsed the file 'D:\MMOS2\INSTALL\BASE.SCR'.
Starting to parse the file 'D:\MMOS2\INSTALL\SMVINI.SCR'.
Successfully parsed the file 'D:\MMOS2\INSTALL\SMVINI.SCR'.
Starting to parse the file 'D:\MMOS2\INSTALL\BASECONF.CH'.
Successfully parsed the file 'D:\MMOS2\INSTALL\BASECONF.CH'.
Starting to parse the file 'D:\MMOS2\INSTALL\SMVCONF.CH'.
Successfully parsed the file 'D:\MMOS2\INSTALL\SMVCONF.CH'.
Loaded DLL D:\MMOS2\DLL\ITERM.DLL and calling entry point ITermEntry.
Returned from DLL D:\MMOS2\DLL\ITERM.DLL.
Loaded DLL D:\MMOS2\DLL\WEPMPINT.DLL and calling entry point WepmPlusAdd.
Returned from DLL D:\MMOS2\DLL\WEPMPINT.DLL.
```

Real-Time MIDI Subsystem

The real-time MIDI subsystem (RTMIDI) provides support for handling real-time MIDI operations for applications.

To provide a robust system for real-time MIDI processing, RTMIDI uses an object-oriented paradigm. *Nodes* are entities that can send or receive MIDI messages, perhaps manipulating the messages before sending them back out again. Nodes that only manipulate messages are called *filter nodes* . Nodes that can generate messages are called *source nodes* .

There are node classes and node instances, where the term node implies *node instance* . In an object-oriented programming language, classes define what variables and operations are available and what they are called. When a class is instantiated, an object that has a physical existence is created. RTMIDI operates the same way.

When a node wants to be "known" by RTMIDI, it must first register a *node class* . This step is only necessary for user-defined classes, since the pre-defined classes are registered by default. Registration provides two pieces of information for RTMIDI: the name of the class and the addresses of all the requisite functions. For example, RTMIDI will need to know what function to call to send a message to the node.

Once a class is registered, it can be instantiated. And once instantiated, it can be connected to other instances. The collection of instances and links is known as the *node network* . Each instance of a class also has its own instance data. When an instance is created, RTMIDI allocates a block of memory. Whenever any of the functions for this class are called, the instance data pointer is passed. This allows a particular function to determine which instance is being called and also allows it to store instance-specific data.

Instances are sometimes called nodes, especially if they are qualified by their class. For example, a hardware node is an instance of the hardware class.

Node Classes

Node classes are descriptions of nodes. From the application's (and user's) point of view, classes are abstractions that have only three features:

- *They have a name.* This name is stored in the MIDI driver itself, and can be used by applications or the user to identify the class. However, the driver itself uses numbers to identify classes, so the applications (or the user) must provide a mapping from class names to class numbers.
- *They can be instantiated.* A node instance of that class is then created. See Node Instances for the features of a node instance.
- *They define the interpretation of the compound message.* The architecture itself doesn't care about the contents of the

compound messages. This is the responsibility of the classes. The compound message format described above is the one used by all the pre-defined classes. It is possible to create a collection of classes which uses a different format, provided the size of the structure is the same.

Generally, all instances of a particular class perform the same function, but this is not a requirement. If a class supports it, the application can send data to a particular instance of that class at any time. This data can be used to configure (or reconfigure) the instance, possibly giving it completely different functionality.

Node Instances

Node instances are physical manifestations of node classes. A node instance has the following functions:

- *It can generate messages.* An instance can create a compound message. For example, an instance of the hardware class can create a message from a stream of bytes sent by a sound card. The hardware instance collects bytes until an entire compound message arrives. At this point, it has a complete compound message ready for further processing.

- *It can be linked to another instance.* Compound messages travel from one instance to another. The link tells the instance where to send the message, if it decides to send it at all.

- *It can send a message to another instance.* The link tells the instance where to send the message. Note that the instance can decide at run-time whether to send the message. For example, an instance that performs filtering can decide to forward messages that fit a certain criteria. Messages that are not forwarded are lost.

- *It can be configured.* An application can send configuration data to a particular instance at any time. The class determines whether configuration data is accepted, but the instance processes the data. Typically, the configuration data changes some of the parameters of the instance. For example, the filter instance (mentioned in the previous list item) can accept data that changes the filter criteria. Another example is an instance that can accept actual machine language instructions and execute them whenever a message arrives. This feature would make that class very powerful, and very dangerous.

- *It can be identified with a unique number.* This 32-bit number is called the instance number and it is used internally as a 16:16 pointer to the instance data. When another driver or an application wishes to reference a particular instance, it uses the instance number.

RTMIDI comes with pre-defined classes. Each of these is discussed in detail in Pre-defined Classes.

The Scheduler

The scheduler is the mechanism that passes messages from one instance to another. It allows multiple instances to pass messages "simultaneously" to multiple targets. The scheduler uses two round-robin queues, Q1 and Q2. Q1 is for real-time messages, and Q2 for non-real-time System Exclusive (SysEx) messages. The scheduler uses the following algorithm:

1. Some instance creates a message. This can occur during interrupt time (for example, from a hardware node) or via task time (for example, from an application node). It adds that message to the end of either Q1 or Q2. It then instructs the scheduler to process the queues.

2. The scheduler runs. If an interrupt from a Type A driver occurs, it is possible for a hardware node to create a new message while the scheduler is running. (See the *MMPM/2 Device Driver Reference* for a description of a Type A driver.) The scheduler can accept new entries into either queue while it is running. It will disable interrupts only during those times when the queue is manipulated.

3. The scheduler first processes all entries in Q1.

4. When Q1 is empty, the scheduler looks at Q2. If it is empty, the scheduler exits. Otherwise, it processes only one element in Q2. Steps 3 and 4 are repeated.

In this manner, real-time messages in Q1 are given priority over messages in Q2. Source nodes should put non-real-time SysEx messages into Q2, and all other messages in Q1. This prevents bulk SysEx dumps from blocking the real-time messages.

The scheduler has the ability to use a *scheduler daemon* to run. After a source instance places a message on a scheduler queue, it tells the scheduler to run. However, this might occur at interrupt time if the message is from a hardware node. If the scheduler were to run at interrupt time as well, system performance could be significantly reduced.

The desired effect would be to run the scheduler in task time, after the interrupt handler has exited. This is known as *deferred interrupt processing* and the scheduler daemon provides this. The daemon is a small program which effectively creates a ring 0 super high priority thread. These threads are guaranteed to run immediately after the current thread or context has completed. In other words, if an interrupt handler unblocks such a thread, that thread is run immediately after the interrupt handler exits. If it is unblocked by a ring 3 process, then it preempts the process that unblocked and runs immediately. If it is unblocked by another ring 0 task thread, it runs after that thread has returned to ring 3 (for example, after the strategy call).

When the scheduler is asked to process the queues, it checks to see whether it is running in another thread. If not, then it processes the queues immediately. Otherwise, it checks to see if the scheduler daemon is available. If so, then it unblocks the daemon thread, which will eventually run the scheduler. If there is no daemon, then it has no choice but to process the queues immediately. If the node network is complex, then the scheduler could spend a significant amount of time in the interrupt handler.

Pre-defined Classes

See the following sections for descriptions of the pre-defined classes that are provided with RTMIDI.

The Hardware Class

Each MIDI port or other hardware that can receive or transmit MIDI data is a Type A device. A Type A device is supported by one Type A driver, but a Type A driver can support any number of Type A devices. The Type A devices belong to the Type A driver. This means that if a card has two MIDI ports, and one driver is used to support both ports, that driver will belong to two hardware nodes.

The hardware class is used to represent Type A devices. In order for a Type A device to be recognized by RTMIDI, its Type A driver must register the device with RTMIDI. For each device registered, an instance of the hardware class is created.

A hardware node communicates with the Type A driver via inter-device-driver communication, or IDC calls. These are simple far (16:16) calls made from one driver to another, and the calling convention is defined by the drivers. (Inter-device-driver communication is described in detail in the *Physical Device Driver Reference* .)

The MIDI bytes from any messages that are sent to the hardware node are immediately transmitted to the Type A driver, and any data sent from the driver to RTMIDI is packaged into compound messages and forwarded.

When a hardware node is enabled, it tells the corresponding driver to open. This usually means attaching the IRQ and initializing the hardware.

RTMIDI supports the concept of instance names. During Type A registration, the Type A driver tells RTMIDI what the name of the Type A device is. This is typically more descriptive than the eight-byte name stored in the Type A driver's device header. For instance, the MPU-401 driver's device name is "MPU401$", but its instance name is "MPU-401 #1 (I/O=0330, IRQ=05)". If the name already exists, an error is returned, and the driver can either pick another name and try again, or fail to load.

The Application Class

This class provides a link between applications and RTMIDI. Just like the IDC interface is the bridge between the hardware nodes and the Type A drivers, the IOCtl interface is the link between application nodes and applications. Only Type A drivers can create and destroy hardware nodes, and only applications can create and destroy application nodes.

Applications communicate with application nodes using the MIDISendMessage functiion, which is used to send a block of compound messages. Each compound message has a time stamp telling RTMIDI when the message is to be sent. When the application sends a block of messages, the application node checks the timestamps of all the messages. Each message that has a timestamp equal to or less than the current time is processed immediately. The remaining messages are queued. If the queue is not large enough to hold all messages, the driver places as many messages as it can into the queue and blocks the thread. As time passes, messages are removed from the queue. Eventually, the remaining messages will be queued and at this point the thread is unblocked.

The timestamps of all compound messages in a block must be in chronological order. If a particular message in the block has a timestamp value less than the previous message (msg[n].ulTime < msg[n-1].ulTime), then the timestamp will be treated as if it were equal to the previous message's timestamp. To send a message immediately, set its timestamp to zero.

See the *Multimedia Programming Reference* for documentation of the RTMIDI application programming interface (API).

Stream Handler Module Definitions

The following information describes the high-level design and operation of the stream handlers provided with the OS/2 multimedia system. The handlers described include:

- Audio Stream Handler DLL and "Stub" Device Driver
- MIDI Mapper Stream Handler DLL
- File System Stream Handler DLL
- Memory Stream Handler DLL
- CD-DA Stream Handler DLL
- CD-ROM XA Stream Handler DLL

Audio Stream Handler

The Audio Stream Handler is used to control streaming of audio data. In some cases, the Audio Stream Handler acts as a consumer of audio data, and in other cases it acts as a producer. The audio data in each of these cases is always digital. It is sometimes compressed and sometimes uncompressed. Each of these cases is discussed in detail below. In addition to these discussions, information is included concerning interaction between this module and the Sync/Stream Manager.

This module is implemented as a DLL running at Ring 3 as well as a stub device driver module (PDD) running at Ring 0 (system kernel privilege level).

The Audio Stream Handler running at Ring 3 can call Ring 3 DLL audio CODECs to assist in compressing and decompressing audio data. The Ring 3 Audio Stream Handler communicates with audio device drivers through the DDCMD interfaces in the VSD. (Refer to the *OS/2 Multimedia Programming Reference* for the VSD DDCMD interface definitions.)

Existing audio device drivers use standard audio-device-driver interface (DDCMDs) to communicate with the Ring 0 stub device driver module.

External Interface Description

The description for the Ring 3 Audio Stream Handler external interface follows:

File Name	AUDIOSH.DLL
Handler Name	AUDIOSH$
Handler Class	AUDIOSH
DD/DLL	DLL
Source	This stream handler can be a source when recording audio data from an audio device. When using this component to record digitized audio, it is known as an audio *Source Stream Handler*. Audio signals are brought into the system and digitized by an audio hardware device. The resultant data is then passed to the consumer stream handler by way of a set of stream buffers. The output of this stream handler can be in several different forms such as uncompressed 16-bit PCM, and several styles of compressed ADPCM.
Target	This stream handler can be a target when playing audio data from a source. When using this stream handler to play back recorded audio samples, it is acting as an audio *target stream handler*. Digitized audio data will be streamed from the *source stream handler* (for example, the File System Stream Handler) through the Sync/Stream Manager. This data is transferred through a number of stream buffers, which are allocated and managed through the

Sync/Stream Manager.

The exact number and size of these stream buffers will depend on the type of data being streamed. The information needed to determine the size and number of stream buffers is contained in the *stream protocol control block* (SPCB) for a particular type of stream.

This stream handler supports audio data in several different formats (stream types) such as 16-bit PCM, AVC's 8-bit ADPCM, CD-ROM XA's 8-bit ADPCM, and so forth.

In addition to handling the playback of audio data, this stream handler generates sync pulses based on the playback of an audio stream. Therefore, this stream handler can be a master or a slave in a sync relationship. The application can set up a sync relationship between two streams and define the sync pulse granularity. Sync pulses would then be generated by the Audio Stream Handler, possibly on a recurring interval, based on the sync granularity defined in the SPCB for the data type.

Device Control Blocks

The Ring 3 Audio Stream Handler supports two *device control blocks* (DCBs). The DCB is used to associate an audio device driver with this stream handler for this stream instance. The DCB is passed as a parameter on a call to SpiCreateStream.

```
 /*****************************************
  *
  * DCB_AUDIOSH - Device Control Block for the
  *               Audio Stream Handler.
  *               (*** PACKED STRUCTURE ***)
  *****************************************/
 typedef struct _dcb_audiosh { /* dcb_audiosh  -  Device Control Block */
    ULONG   ulDCBLen;                /* Length of structure          */
    SZ      szDevName[MAX_SPI_NAME];   /* Device driver name          */
    ULONG   ulSysFileNum;            /* File handle number - From an */
                                     /* audio_init call              */
    } DCB_AUDIOSH;
 typedef struct _dcb_audiosh FAR  *PDCB_AUDIOSH;
```

The Ring 3 Audio Stream Handler also supports the VSD device control block (VSD_DCB) which identifies the VSD DLL to be used by the Ring 3 Audio Stream Handler. The VSD_DCB is passed as a parameter on a call to SpiCreateStream.

```
/*************************************************************************
 *
 * VSD_DCB - VSD Device Control Block
 *
 *         This structure will allow stream handlers to use the VSD DLL
 *         by using by the additional fields in the structure.
 *
 *               (*** PACKED STRUCTURE ***)
 *************************************************************************/
typedef struct _VSD_DCB { /* vsd_dcb - VSD Device Control Block           */

   ULONG   ulDCBLen;               /* Length of structure               */
   SZ      szDevName[MAX_SPI_NAME]; /* Device driver name        */
   ULONG   ulSysFileNum;           /* File handle number                */
   ULONG   hvsd;                   /* Handle to VSD instance            */
   PFN     pfnvsdEntryPoint;       /* Address of VSD entry point        */
   ULONG   ulReserved1;            /* Reserved for system               */
   ULONG   ulReserved2;            /* Reserved for system               */

   } VSD_DCB;

typedef VSD_DCB FAR *PVSD_DCB;
```

Associate Control Blocks

The Audio Stream Handler does not support an associate control block.

Implicit Events (EVENT_IMPLICIT_TYPE) Supported

The following implicit (EVENT_IMPLICIT_TYPE) events for the Audio Stream Handler are supported:

- EVENT_ERROR

  The *ulStatus* field will contain the error code. The possible error codes that can be generated and returned by this stream handler are:
  - ERROR_INVALID_BUFFER_RETURNED
  - ERROR_DEVICE_OVERRUN
  - ERROR_STREAM_STOP_PENDING
- EVENT_PLAYLISTCUEPOINT

  This will be generated only when this stream handler is a target and it finds a cuepoint indicator in the buffer table from the Sync/Stream Manager. The event will be reported after the data has been written to the file system using MMIO. The *ulMessageParm* field of the PLAYL_EVCB will be filled in with the message supplied in the playlist instruction. The *mmtimeStream* field will not be filled in.

Explicit Events Supported

The following explicit events for the Audio Stream Handler are supported:

- EVENT_CUE_TIME
- EVENT_CUE_TIME_PAUSE
- EVENT_DATAUNDERRUN

Explicit Events Not Supported

The following explicit events for the Audio Stream Handler are not supported:

- EVENT_CUE_DATA
- EVENT_DATAOVERRUN (Not Supported)

Stream Handler Commands Supported

The following stream handler commands (SHC) are supported. Refer to the *OS/2 Multimedia Programming Reference* for a description of these SHC commands and the error return codes.

- SHC_ASSOCIATE

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_REQUEST

- SHC_CLOSE

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION

- SHC_CREATE

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_STREAM_CREATION
  - ERROR_INVALID_STREAM
  - ERROR_DEVICE_NOT_FOUND
  - ERROR_INVALID_BLOCK
  - ERROR_INVALID_SPCBKEY

- SHC_DESTROY

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM

- SHC_DISABLE_EVENT

  Possible Return Codes:

- NO_ERROR
- ERROR_INVALID_FUNCTION
- ERROR_INVALID_EVENT

- SHC_DISABLE_SYNC

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION

- SHC_ENABLE_EVENT

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_TOO_MANY_EVENTS
  - ERROR_INVALID_EVENT

- SHC_ENABLE_SYNC

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_FLAG
  - ERROR_INVALID_STREAM
  - ERROR_STREAM_NOTMASTER
  - ERROR_INVALID_NUMSLAVES

- SHC_ENUMERATE_PROTOCOLS

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INSUFF_BUFFER

- SHC_GET_PROTOCOL

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_SPCBKEY

- SHC_GET_TIME

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM

- SHC_INSTALL_PROTOCOL

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

- SHC_NEGOTIATE_RESULT

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION

- SHC_SEEK

Possible Return Codes:

- NO_ERROR
- ERROR_INVALID_FUNCTION
- ERROR_STREAM_NOT_SEEKABLE
- ERROR_DATA_ITEM_NOT_SEEKABLE

- SHC_START

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM

- SHC_STOP

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM

Base Stream Protocol Control Blocks Supported

The Audio Stream Handler supports the following SPCBs:

- DATATYPE_WAVEFORM

  Subtypes:

  - WAVE_FORMAT_1M08
  - WAVE_FORMAT_1S08
  - WAVE_FORMAT_1M16
  - WAVE_FORMAT_1S16
  - WAVE_FORMAT_2M08
  - WAVE_FORMAT_2S08
  - WAVE_FORMAT_2M16
  - WAVE_FORMAT_2S16
  - WAVE_FORMAT_4M08
  - WAVE_FORMAT_4S08
  - WAVE_FORMAT_4M16
  - WAVE_FORMAT_4S16
  - WAVE_FORMAT_8M08
  - WAVE_FORMAT_8S08
  - WAVE_FORMAT_8M16
  - WAVE_FORMAT_8S16

- DATATYPE_ALAW

  Subtypes:

  - ALAW_8B8KM
  - ALAW_8B11KM
  - ALAW_8B22KM
  - ALAW_8B44KM
  - ALAW_8B8KS
  - ALAW_8B11KS
  - ALAW_8B22KS
  - ALAW_8B44KS

- DATATYPE_MULAW

  Subtypes:

  - MULAW_8B8KM
  - MULAW_8B11KM
  - MULAW_8B22KM
  - MULAW_8B44KM
  - MULAW_8B8KS
  - MULAW_8B11KS
  - MULAW_8B22KS
  - MULAW_8B44KS

- DATATYPE_ADPCM_AVC

  Subtypes:

  - ADPCM_AVC_VOICE
  - ADPCM_AVC_MUSIC
  - ADPCM_AVC_STEREO
  - ADPCM_AVC_HQ

- DATATYPE_MIDI

  Subtypes:

  - SUBTYPE_NONE

- DATATYPE_SPV2

  Subtypes:

  - SPV2_BPCM
  - SPV2_PCM
  - SPV2_NONE

Stream Handler Limits

The Audio Stream Handler is limited by system memory.

MIDI Mapper Stream Handler

The MIDI Mapper Stream Handler is used to map MIDI data. This stream handler does not interface with an audio device driver directly but is basically a "filter" stream handler that filters the MIDI data. The Audio Stream Handler is used to interface to the audio device drivers.

This module is implemented as an OS/2 DLL running at ring 3.

Flushing a Filter Stream Group

A filter stream group requires some extra steps to properly erase its contents. Stop flushes must be sent separately to each stream. An example stream group of one master stream connected through a filter handler to a slave stream would need a stop flush sent first to the master stream. When the master stop event is received, the second stop flush must then be sent to the slave stream. Refer to the SHC_STOP command message in the *OS/2 Multimedia Programming Reference* .

Application and Media Driver Capabilities

For optimum performance, each application and media driver should have the following capabilities:

SEEK	Seeks the source stream and then the target streams. (The stream time will not be correct if you use another other method.)
STOP DISCARD	Stops the target streams and waits for all stop events. Next, it stops the source stream and waits for the event. If a stream is at EOS, an error stream that is already stopped will be returned. Failure to wait for events on the source stream will deadlock.
STOP FLUSH	Stops the source stream and waits for the stop event. Next, it stops the target streams and waits for the stop events. Failure to wait for events on the source stream will cause deadlock.
STOP PAUSE	Always pauses all streams at the same time. Pausing one stream while leaving another running can cause deadlock.
START	Starts the source stream and then starts the target streams. Small MIDI files will produce a quick EOS on the source stream. Do not attempt to START or START PREROLL a target stream without first starting the source stream. An attempt to do this will deadlock.
START PREROLL	Starts (nonpreroll) the source stream and then starts preroll target streams. If

	you attempt to START PREROLL all streams, you will deadlock.
DESTROY	Destroys all streams at the same time. Do not try to destroy one stream and then continue to use another stream.
CREATE and ASSOCIATE	All streams must be created and associated before any streams can accept commands. Do not attempt to add another stream later on.

External Interface Description

The description for the MIDI Mapper Stream Handler external interface follows:

File Name	MISH.DLL
Handler Name	MISH
Handler Class	MIDISYS
PDD/DLL	DLL
Source and Target	This stream handler is a filter. Therefore, it is both a source and a target stream handler at the same time. It consumes MIDI data from the source stream and produces mapped MIDI data that goes into the target stream or streams. The MIDI Mapper Stream Handler can have one source stream (the master stream). This stream handler does not generate or receive sync pulses, but it can be included in a sync group. In fact, MIDI mapping is done by grouping the input stream (master stream) with the output streams (slave streams) to create a sync group. A stream will be created for each output port. The MIDI sync group can be started, stopped, and seeked as a group by using the "slaves" flag with each of these calls.

Device Control Blocks

The MIDI Mapper Stream Handler does not support a device control block.

Associate Control Blocks

The MIDI Mapper Stream Handler supports the following associate control blocks shown in the following figure.

```
 /***************************************************
 * MISH - MIDI stream handler port-stream table ACB
 ***************************************************/
 #define ACBTYPE_MISH       0x0005L   /* MIDI port-stream table        */
 typedef struct _acb_MISH     /* acbmish - MIDI Assoc. Control Block   */
    {
    ULONG     ulACBLen;               /* Length of structure           */
    ULONG     ulObjType;              /* ACB_MISH                      */
    HSTREAM   hstreamDefault;         /* Default hstream to use when   */
                                      /*  mapper is turned off.        */
    ULONG     ulReserved1;
    ULONG     ulReserved2;
    ULONG     ulNumInStreams;
    HSTREAM   hstreamInMAX[_PORTS];   /* Array of Input streams        */
    ULONG     ulNumOutStreams;
    HSTREAM   hstreamOut[MAX_PORTS];  /* Array of Output streams       */
                                      /* The index into the array is   */
                                      /*  the source channel for that  */
                                      /*  stream.                      */
    } ACB_MISH;


/***************************************************
 * MISH - MIDI stream handler SET ACB
 ***************************************************/
 #define ACBTYPE_SET       0x0006L    /* MIDI set function            */
 typedef struct _acb_set      /* acbset - Set Assoc. Control Block     */
    {
    ULONG     ulACBLen;               /* Length of structure           */
    ULONG     ulObjType;              /* ACB_SET                       */
    ULONG     ulFlag;                 /* Set flag                      */
    ULONG     ulTempo;                /* Not used.                     */
    } ACB_SET;
```

```
/* ulFlag defines:   */
#define MIDI_MAP_ON        0x0000L   /* turn mapping function on      */
#define MIDI_MAP_OFF       0x0001L   /* turn mapping function off     */
```

Implicit Events (EVENT_IMPLICIT_TYPE) Supported

The following implicit (EVENT_IMPLICIT_TYPE) events for the MIDI Mapper Stream Handler are supported:

- EVENT_ERROR

  The *ulStatus* field will contain the error code. The possible error codes that can be generated and returned by this stream handler are:
  - ERROR_INVALID_BUFFER_RETURNED
  - ERROR_DEVICE_OVERRUN
  - ERROR_STREAM_STOP_PENDING

Explicit Events Supported

No explicit events are supported for the MIDI Mapper Stream Handler.

Stream Handler Commands Supported

The following stream handler commands (SHC) are supported. Refer to the *OS/2 Multimedia Programming Reference* for a description of these SHC commands and the error return codes.

- SHC_ASSOCIATE

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_INVALID_OBJTYPE (only ACBTYPE_MMIO is supported)
  - ERROR_INVALID_BUFFER_SIZE (*ulAcbLen* is smaller than needed)
  - ERROR_STREAM_NOT_STOP (stream must be stopped to associate)

- SHC_CLOSE

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION

- SHC_CREATE

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_STREAM_CREATION
  - ERROR_INVALID_STREAM
  - ERROR_DEVICE_NOT_FOUND
  - ERROR_INVALID_BLOCK
  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

- SHC_DESTROY

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION

- SHC_ENUMERATE_PROTOCOLS

Possible Return Codes:

- NO_ERROR
- ERROR_INVALID_FUNCTION
- ERROR_INSUFF_BUFFER

- SHC_GET_PROTOCOL

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_SPCBKEY

- SHC_INSTALL_PROTOCOL

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

- SHC_NEGOTIATE_RESULT

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)

- SHC_SEEK

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_STREAM_NOT_SEEKABLE
  - ERROR_DATA_ITEM_NOT_SEEKABLE
  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)
  - ERROR_NOT_SEEKABLE_BY_TIME
  - ERROR_STREAM_NOT_STOP

- SHC_START

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)
  - ERROR_DATA_ITEM_NOT_SPECIFIED (stream must be associated)

- SHC_STOP

  Possible Return Codes:

  - NO_ERROR
  - ERROR_INVALID_FUNCTION
  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)
  - ERROR_STREAM_NOT_STARTED

Base Stream Protocol Control Blocks Supported

The MIDI Mapper Stream Handler supports the following stream protocol control blocks (SPCBs).

- DATATYPE_MIDI

Stream Handler Limits

Limited by system memory.

File System Stream Handler

The File System Stream Handler DLL transports data to or from file system devices (local or remote) on behalf of a real-time application. This handler is unique in that it utilizes the MMIO subsystem to interface to a very wide variety of devices, such as hard disk drives, diskette drives, CD-ROM drives, WORM drives, and so forth. These devices can be physically installed in the local system hardware, or they can be accessed across a LAN on a server machine.

In a playback scenario (for example, waveform audio from a RIFF file), the File System Stream Handler uses MMIO to perform I/O on specified data files, and then performs stream processing to maintain a continuously available supply which is then streamed to a target stream handler; for example, the Waveform Audio Stream Handler. This handler does not operate fully in a real-time mode, but it must support continuous data streaming. It also does not support synchronization mastering, because the file system devices are not real-time devices.

External Interface Description

The description for the File System Stream Handler external interface follows:

File Name	FSSH.DLL
Handler Name	FSSH
Handler Class	FILESYS
PDD/DLL	DLL
Source	This stream handler *can* be the source in a stream.
Target	This stream handler *can* be the target in a stream.

Device Control Blocks

None.

Associate Control Blocks

This handler supports type ACBTYPE_MMIO associate control blocks.

```
/***************************************************
 * FSSH - File System Stream Handler MMIO Object ACB
 ***************************************************/
 #define ACBTYPE_MMIO      0x0001L       /* MMIO object             */

 typedef struct _acb_mmio               /* acbmmio  - MMIO ACB     */
    {
    ULONG   ulACBLen;                   /* Length of structure     */
    ULONG   ulObjType;                  /* ACB_MMIO                */
    HMMIO   hmmio;                      /* Handle of MMIO object   */
    } ACB_MMIO;
```

Implicit (EVENT_IMPLICIT_TYPE) Events Supported

The following implicit (EVENT_IMPLICIT_TYPE) events for the File System Stream Handler are supported:

- EVENT_ERROR

  The error type will be set in the *ulFlag* field of the event control block. The three types of errors that will be reported are:

  - Temporary Error - TEMPORARY_ERROR 0x0000L

    An error occurred during streaming but the stream handler, the SSM, or both the stream handler and the SSM were able to continue streaming.

  - Recoverable Error - RECOVERABLE_ERROR 0x0001L

    An error occurred that required the stream to be stopped. The application can restart the stream if appropriate.

- NonRecoverable Error - NONRECOVERABLE_ERROR 0x0002L

  A severe error occurred causing the stream handler to stop this stream. The stream cannot be restarted. The application must issue a call to SpiDestroyStream.

The *ulStatus* field will contain the error code. No error codes are generated and returned by this stream handler.

Also, errors from the following APIs can be returned:

- DosGetInfoBlocks
- SMHEntryPoint SMH_NOTIFY
- mmioRead (If source, extended error code obtained by mmioGetLastError)
- mmioWrite (If target, extended error code obtained by mmioGetLastError)

- EVENT_PLAYLISTCUEPOINT

  This will be generated only when this stream handler is a target and it finds a cuepoint indicator in the buffer table from the Sync/Stream Manager. The event will be reported after the data has been written to the file system using MMIO. The *ulMessageParm* field of the PLAYL_EVCB will be filled in with the message supplied in the playlist instruction. The *mmtimeStream* field will not be filled in.

Explicit Events Supported

No explicit events are supported for the File System Stream Handler.

Stream Handler Commands Supported

The following stream handler commands (SHCs) are supported. Refer to the *OS/2 Multimedia Programming Reference* for a description of these SHC commands and the error return codes.

- SHC_ASSOCIATE

  Note: A stream requires a file to be opened and associated to the stream before a stream can be started. Reassociating a new object without stopping the stream is not supported.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_INVALID_OBJTYPE (only ACBTYPE_MMIO is supported)
  - ERROR_INVALID_BUFFER_SIZE (*ulAcbLen* is smaller than needed)
  - ERROR_STREAM_NOT_STOP (stream must be stopped to associate)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_CREATE

  Note: If the *spcbkey* passed does not match any of the installed protocols, the last installed stream protocol control block with DATATYPE_GENERIC will be used.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosCreateThread

- SHC_DESTROY

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or both *hstream* and *hid* passed)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_START

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or both *hstream* and *hid* passed)
  - ERROR_DATA_ITEM_NOT_SPECIFIED (stream must be associated before it is started).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_STOP

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or both *hstream* and *hid* passed)
  - ERROR_STREAM_NOT_STARTED

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosResumeThread
  - DosSuspendThread

- SHC_SEEK

  Note: If you are adding support in the system for a new nonlinear data type, you can write a MMIO IOProc to support the MMIOM_SEEKBYTIME message. This message will be sent (mmioSendMessage) when the File System Stream Handler gets a seek by time (for information on the SpiSeekStream SPI_SEEKMMTIME parameter, refer to the *OS/2 Multimedia Programming Reference* .) and the SPCB indicates this is a nonlinear data type by having a 0 in the spcb.ulBytesPerUnit field, the spcb.mmtimePerUnit fields, or both fields.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or both *hstream* and *hid* passed)
  - ERROR_NOT_SEEKABLE_BY_TIME
  - ERROR_DATA_ITEM_NOT_SEEKABLE
  - ERROR_DATA_ITEM_NOT_SPECIFIED
  - ERROR_STREAM_NOT_STOP

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - mmioSeek (the extended error code from mmioGetLastError)
  - mmioSendMessage - MMIOM_SEEKBYTIME (the extended error code from mmioGetLastError)

- SHC_GET_PROTOCOL

  Possible return codes:

  - ERROR_INVALID_SPCBKEY

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_INSTALL_PROTOCOL

  Note: This stream handler allows any data type and subtype to be installed.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_ENUMERATE_PROTOCOLS

  Possible return codes:

- ERROR_INSUFF_BUFFER

Return codes from the following APIs are also returned:

- DosRequestMutexSem

- SHC_NEGOTIATE_RESULT

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or both *hstream* and *hid* passed)
  - ERROR_INVALID_FUNCTION (must only be called directly after create)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

Base Stream Protocol Control Blocks Data Types Supported

The File System Stream Handler has only 1 base SPCB. It is DATATYPE_GENERIC. When a stream is created with a *SPCBKEY* that is not installed, this stream handler copies over the last installed SPCB of type DATATYPE_GENERIC and uses it. The data type, subtype and *intkey* passed are used in place of the generic values for these fields. The base generic SPCB has:

SPCB Field	DATATYPE_GENERIC Values
ulBufSize	16KB
ulMinBuf	2
ulMaxBuf	5
ulSrcStart	1
ulTgtStart	1
ulBufFlag	SPCBBUF_NONCONTIGUOUS
ulHandFlag	SPCBHAND_NOSYNC | SPCB_PHYS_SEEK

Stream Handler Limits

Maximum number of streams (only limited by available memory).

Memory Stream Handler

There are many multimedia application scenarios where data streaming might be performed to or from system memory. For example, an application can create a waveform data object dynamically, allocating application data buffers in which to store the waveform data. This data is not read from a device but is generated by an application algorithm. This waveform data can then be streamed to the Waveform Audio Stream Handler (target).

Another example where system memory is used for streaming is when the application must process or convert data read from a file before it can be used as the source of a stream. There can be some cases where application-unique data compression is used. In these cases, the application can stream data from a "flat file" into an application buffer. Then, after decompression (or other operation) this data in system memory can be streamed using the Memory Stream Handler as the source, perhaps sending the data to the Waveform Audio Stream Handler.

The Memory Stream Handler does not support synchronization.

Memory playlists are used to specify the memory addresses to play from or record into.

External Interface Description

The description for the Memory Stream Handler external interface follows:

File Name	MEMSH.DLL
Handler Name	MEMSH
Handler Class	MEMSYS
PDD/DLL	DLL
Source	This stream handler *can* be the source in a stream.
Target	This stream handler *can* be the target in a stream.

Device Control Blocks

None.

Associate Control Blocks

This handler supports type ACBTYPE_MEM_PLAYL associate control blocks.

```
/************************************************
 * MEMSH - Memory Stream Handler Playlist Object ACB
 ************************************************/
#define ACBTYPE_MEM_PLAYL     0x0003L  /* Memory playlist object     */

typedef struct _acb_mem_playl
   {
   ULONG   ulACBLen;              /* Length of structure            */
   ULONG   ulObjType;             /* ACBTYPE_MEM_PLAYL              */
   PVOID   pMemoryAddr;           /* Starting address of playlist   */
   } ACB_MEM_PLAYL;
```

Implicit (EVENT_IMPLICIT_TYPE) Events Supported

The following implicit (EVENT_IMPLICIT_TYPE) events for the Memory Stream Handler are supported:

- EVENT_ERROR

  The error type will be set in the *ulFlag* field of the event implicit control block (EVCB). These error types and the flag values are described in the file system stream handler section.

  The *ulStatus* field will contain the error code. The possible error codes that can be generated and returned by this stream handler are:

  - ERROR_PLAYLIST_STACK_OVERFLOW (too many CALL instructions encountered. The maximum CALL depth is 20 calls.)
  - ERROR_PLAYLIST_STACK_UNDERFLOW (too many RETURN instructions encountered. There should be exactly one RETURN instruction executed for each CALL instruction.)
  - ERROR_INVALID_FUNCTION (invalid playlist opcode encountered.)
  - ERROR_INVALID_BLOCK (playlist is not read/write accessible or data is not read (play) or write (record) accessible.)
  - ERROR_END_OF_PLAYLIST (when recording into a playlist, the EXIT opcode was encountered before EOS. The stream handler stops).

  Also, errors from the following APIs can be returned:

  - DosGetInfoBlocks
  - DosQueryMem
  - SMHEntryPoint SMH_NOTIFY (GiveBuf function for Play)
  - SMHEntryPoint SMH_NOTIFY (GetFull, ReturnEmpty for Record)
  - SMHEntryPoint SMH_REPORTEVENT

- EVENT_PLAYLISTCUEPOINT

  The Memory Stream Handler does not receive playlist cuepoint indicators within the stream. This handler will place a cuepoint indicator into a stream when it interprets a CUEPOINT playlist operation. Therefore this stream handler does not generate the EVENT_PLAYLISTCUEPOINT, but it enables it in the stream for the target stream handler. Check the description of the target

stream handler to determine if it supports this EVENT_PLAYLISTCUEPOINT.

- EVENT_PLAYLISTMESSAGE

  This is generated when a MESSAGE instruction is encountered. The *ulMessageParm* field of the PLAYL_EVCB will be filled in with the message supplied in the message parameter (operand2) of the playlist MESSAGE instruction. The *ulStatus* field contains the playlist instruction number.

Explicit Events Supported

No explicit events are supported for the Memory Stream Handler.

Stream Handler Commands Supported

The following stream handler commands (SHC) are supported. Refer to the *OS/2 Multimedia Programming Reference* for a description of these SHC commands and the error return codes.

- SHC_ASSOCIATE

  Note: A stream requires a file to be opened and associated to the stream before a stream can be started. Reassociating a new object without stopping the stream is not supported.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)
  - ERROR_INVALID_OBJTYPE (only ACBTYPE_MEM_PLAYL is supported)
  - ERROR_INVALID_BUFFER_SIZE (*ulAcbLen* is smaller than needed)
  - ERROR_STREAM_NOT_STOP (stream must be stopped to associate)
  - ERROR_INVALID_BLOCK (invalid memory address - not read/write accessible)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosQueryMem

- SHC_CREATE

  Note: If the *SPCBKEY* passed does not match any of the installed protocols, the last installed SPCB with DATATYPE_GENERIC will be used. If the MEMSH is the source, create will always set the SPCBBUF_USERPROVIDED flag in the spcb.ulBufFlag field.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosCreateThread

- SHC_DESTROY

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_START

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_DATA_ITEM_NOT_SPECIFIED (stream must be associated before it is started).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_STOP

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid*-or both-passed)
  - ERROR_STREAM_NOT_STARTED

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosResumeThread
  - DosSuspendThread

- SHC_SEEK

  Note: Seeking backward is not supported. SEEK_END is only supported if the *lSeekPoint* is 0. A playlist seek is accomplished by going back to the beginning of the playlist and executing each instruction without playing the data, but counting the time it takes. The Seek operates on the current playlist, that means that if the playlist was modified by the application the Seek uses the modified playlist to calculate the seek position. When a seek absolute is performed all LOOP iterations are reset to 0 before the Seek starts calculating the position.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid*-or both-passed)
  - ERROR_NOT_SEEKABLE_BY_TIME
  - ERROR_DATA_ITEM_NOT_SEEKABLE
  - ERROR_STREAM_NOT_STOP
  - ERROR_SEEK_BACK_NOT_SUPPORTED
  - ERROR_SEEK_PAST_END
  - ERROR_INVALID_BLOCK
  - ERROR_PLAYLIST_STACK_OVERFLOW
  - ERROR_PLAYLIST_STACK_UNDERFLOW
  - ERROR_INVALID_FUNCTION
  - ERROR_DATA_ITEM_NOT_SPECIFIED

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosQueryMem

- SHC_GET_PROTOCOL

  Possible return codes:

  - ERROR_INVALID_SPCBKEY

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_INSTALL_PROTOCOL

  Note: This stream handler allows any data type and subtype to be installed.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_ENUMERATE_PROTOCOLS

  Possible return codes:

  - ERROR_INSUFF_BUFFER

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_NEGOTIATE_RESULT

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_INVALID_FUNCTION (must only be called directly after create)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

Base Stream Protocol Control Block Data Types Supported

The Memory Stream Handler has only 1 base stream protocol control block (SPCB). It is DATATYPE_GENERIC. When a stream is created with a *SPCBKEY* that is not one installed, this stream handler copies over the last installed SPCB of type DATATYPE_GENERIC and uses it. The data type, subtype and *intkey* passed are used in place of the generic values for these fields. The base generic SPCB has:

SPCB Field	DATATYPE_GENERIC Values
ulBufSize	16KB
ulMinBuf	3
ulMaxBuf	10
ulSrcStart	1
ulTgtStart	1
ulBufFlag	SPCBBUF_NONCONTIGUOUS
ulHandFlag	SPCBHAND_NOSYNC | SPCBHAND_PHYS_SEEK

Stream Handler Limits

Maximum number of streams (only limited by available memory).

Compact Disc-Digital Audio Stream Handler

Many CDs have CD-DA ("*Redbook audio* ") tracks that contain digital audio. These audio tracks can be played using the built-in DAC or the data can be read into the system and streamed to an adapter with DAC hardware KB (for example, ACPA card). For the latter case, the CD-DA Stream Handler DLL is required and acts as the source stream handler.

This handler does not operate fully in a real-time mode, but it must support continuous data streaming. It also does not support synchronization mastering, because of the lack of real-time nature of CD devices.

This handler operates on the specific CD addresses. Most CD-DA do not have data in approximately the first 2 seconds of the CD, but some discs do. CD-DA tracks can also be placed between other data tracks on mixed-mode disks. Because of this operation, the CD-DA Stream Handler will interpret the absolute beginning of the disc as MMTIME 0. The calling program must query the disc and find where the desired CD-DA track begins and call SpiSeekStream before starting the stream. The CD Audio media control interface driver does this operation for the applications coding to the media control interface.

External Interface Description

The description for the Compact Disc-Digital Audio Stream Handler external interface follows:

File Name	CDDASH.DLL
Handler Name	CDDASH
Handler Class	FILESYS
PDD/DLL	DLL

Source	This stream handler *can* be the source in a stream.
Target	This stream handler *cannot* be the target in a stream.

Device Control Blocks

None.

Associate Control Blocks

This handler supports type ACBTYPE_CDDA associate control blocks.

```
/*************************************************
 * CDDASH - CD DA Stream Handler Object ACB
 *************************************************/
 #define ACBTYPE_CDDA        0x0004L   /* Compact disc - digital audio */
 typedef struct _acb_CDDA        /* acbcdda - CD Assoc. Control Block  */
    {
    ULONG   ulACBLen;              /* Length of structure     */
    ULONG   ulObjType;             /* ACB_CDDA                */
    CHAR    bCDDrive;              /* CD drive letter         */
    } ACB_CDDA;
```

Implicit (EVENT_IMPLICIT_TYPE) Events Supported

The following implicit (EVENT_IMPLICIT_TYPE) events for the Compact Disc-Digital Audio Stream Handler are supported:

- EVENT_ERROR

  The error type will be set in the *ulFlag* field of the event implicit control block (EVCB). The three types of errors that will be reported are:

  - Temporary Error (TEMPORARY_ERROR 0x0000L)

    An error occurred during streaming but the stream handler, the Sync/Stream Manager or both the stream handler and the Sync/Stream Manager were able to continue streaming.

  - Recoverable Error (RECOVERABLE_ERROR 0x0001L)

    An error occurred that required the stream to be stopped. The application can restart the stream if appropriate.

  - NonRecoverable Error (NONRECOVERABLE_ERROR 0x0002L)

    A severe error occurred causing the stream handler to stop this stream. The stream can not be restarted. The application must issue an SpiDestroyStream.

  The *ulStatus* field will contain the error code. The possible error codes that can be generated and returned by this stream handler are:

  - None

  Also, errors from the following APIs can be returned:

  - DosGetInfoBlocks
  - SMHEntryPoint SMH_NOTIFY
  - DosDevIOCtl (from ReadLong command to CD device driver).

Explicit Events Supported

No explicit events are supported for the Compact Disc-Digital Audio Stream Handler.

Stream Handler Commands Supported

The following stream handler commands (SHC) are supported. Refer to the *OS/2 Multimedia Programming Reference* for a description of these SHC commands and the error return codes.

- SHC_ASSOCIATE

  Note: A stream requires a file to be opened and associated to the stream before a stream can be started. Reassociating a new

drive letter without stopping the stream is not supported.

Possible return codes:

- ERROR_INVALID_STREAM (invalid *hstream* or *hid*-or both-passed)
- ERROR_INVALID_OBJTYPE (only ACBTYPE_CDDA is supported)
- ERROR_INVALID_BUFFER_SIZE (*ulAcbLen* is smaller than needed)
- ERROR_STREAM_NOT_STOP (stream must be stopped to associate)

Return codes from the following APIs are also returned:

- DosRequestMutexSem
- DosOpen (for CD device).

- SHC_CREATE

  Note: The only data type supported is 16-bit PCM Stereo sampled at 44.1 kHz.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosCreateThread

- SHC_DESTROY

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_START

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)
  - ERROR_DATA_ITEM_NOT_SPECIFIED (stream must be associated before it is started).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_STOP

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hstream* and *hid* passed)
  - ERROR_STREAM_NOT_STARTED

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosResumeThread
  - DosSuspendThread

- SHC_SEEK

  Note: Keep in mind that SEEK will go to the specified address in absolute terms. The absolute address from the beginning of the disc might not be the beginning of the CD-DA music.

  Also the seekpoint is granular to the *mmtimePerUnit* specified in the SPCB.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid*-or both-passed)

- ERROR_DATA_ITEM_NOT_SEEKABLE
- ERROR_DATA_ITEM_NOT_SPECIFIED
- ERROR_STREAM_NOT_STOP

- ERROR_SEEK_PAST_END
- ERROR_SEEK_BEFORE_BEGINNING

Return codes from the following APIs are also returned:

- DosRequestMutexSem
- DosDevIOCtl (DiskInfo command for Seek_End)
- DosDevIOCtl (Seek command)

- SHC_GET_PROTOCOL

  Possible return codes:

  - ERROR_INVALID_SPCBKEY

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_INSTALL_PROTOCOL

  Note: This stream handler allows any data type and subtype to be installed.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_ENUMERATE_PROTOCOLS

  Possible return codes:

  - ERROR_INSUFF_BUFFER

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_NEGOTIATE_RESULT

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid*-or both-passed)
  - ERROR_INVALID_FUNCTION (must only be called directly after create).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

Base Stream Protocol Control Block Data Types Supported

The Compact Disc-Digital Audio Stream Handler has only 1 base SPCB. It is DATATYPE_WAVEFORM with DataSubType WAVE_FORMAT_4S16. This SPCB has the following defaults:

SPCB Field	DATATYPE_WAVEFORM Values
ulBufSize	48KB
ulMinBuf	8
ulMaxBuf	12
ulSrcStart	1

```
ulTgtStart          7
ulBufFlag           SPCBBUF_FIXEDBUF
ulHandFlag          SPCBHAND_NOSYNC | SPCBHAND_PHYS_SEEK
ulBytesPerUnit      588
mmtimePerUnit       10
```

Stream Handler Limits

Maximum number of streams (only limited by available memory).

CD-ROM XA Stream Handler

The CD-ROM XA Stream Handler is used to stream interleaved data to multiple streams and therefore multiple target devices. This is required to play multiple streams from a CD-ROM XA. For example, a CD-ROM XA file could have one channel of audio and one channel of video that need to be streamed at the same time. Because the audio and video are interleaved in the same file the CD-ROM XA Stream Handler can take these individual channels from the file and pass them to the Sync/Stream Manager as independent streams.

External Interface Description

The description for the CD-ROM XA Stream Handler external interface follows:

File Name	SSSH.DLL
Handler Name	SSSH
Handler Class	FILESYS
PDD/DLL	DLL
Source	This stream handler *can* be the source in a stream.
Target	This stream handler *cannot* be the target in a stream.

Device Control Blocks

This handler requires the standard DCB structure on the SHC_CREATE function. The *szDevName* must have the drive letter of the CD-ROM XA drive.

```
typedef struct _dcb             /* dcb  -  Device Control Block   */
   {
   ULONG   ulDCBLen;                  /* Length of structure  */
   SZ      szDevName[MAX_SPI_NAME];    /* Device driver name   */
   } DCB;
```

Associate Control Blocks

None.

Implicit (EVENT_IMPLICIT_TYPE) Events Supported

The following implicit (EVENT_IMPLICIT_TYPE) events for the CD-ROM XA Stream Handler are supported:

- EVENT_ERROR

  The error type will be set in the *ulFlag* field of the event implicit control block (EVCB). The three types of errors that will be reported are:

- Temporary Error (TEMPOARY_ERROR 0x0000L)

  An error occurred during streaming but the stream handler, the Sync/Stream Manager or both the stream handler and the Sync/Stream Manager were able to continue streaming.

- Recoverable Error (RECOVERABLE_ERROR 0x0001L)

  An error occurred that required the stream to be stopped. The application can restart the stream if appropriate.

- NonRecoverable Error (NONRECOVERABLE_ERROR 0x0002L)

  A severe error occurred causing the stream handler to stop this stream. The stream cannot be restarted. The application must issue an SpiDestroyStream.

The *ulStatus* field will contain the error code. The possible error codes that can be generated and returned by this stream handler are:

- None

Also, errors from the following APIs can be returned:

- DosRequestMutexSem
- DosGetInfoBlocks
- SMHEntryPoint SMH_NOTIFY
- DosDevIOCtl (from ReadLong command to CD device driver).

Explicit Events Supported

No explicit events are supported for the CD-ROM XA Stream Handler.

Stream Handler Commands Supported

The following stream handler commands (SHC) are supported. Refer to the *OS/2 Multimedia Programming Reference* for a description of these SHC commands and the error return codes.

- SHC_ASSOCIATE

  Note: A stream requires a file to be opened and associated to the stream before a stream can be started.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_INVALID_OBJTYPE (only ACBTYPE_SSSH is supported)
  - ERROR_INVALID_BUFFER_SIZE (*ulAcbLen* is smaller than needed)
  - ERROR_STREAM_NOT_STOP (stream must be stopped to associate)
  - ERROR_FILE_FORMAT_INCORRECT (file is not cdxa mode 2)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosDevIOCtl (ReadLong on CD drive)
  - DosFSCtl (to CDFS)
  - DosSetFilePtr

- SHC_CREATE

  Note: The primary stream is the stream that is created first for a interleaved file. The secondary streams must have the *hstream* handle of the primary stream in the *hstreambuf* parameter of the SpiCreateStream API. Only the CREATE for the primary stream needs to have the DCB filled in with the drive letter. Any DCB passed in on the secondary stream creates is ignored by this stream handler.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES
  - ERROR_TOO_MANY_STREAMS (only 16 streams per XA file)
  - ERROR_INVALID_BUFFER_SIZE (DCB is too small)
  - ERROR_INVALID_PARAMETER (DCB parameter is NULL).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- DosCreateThread
- DosOpen for CD Drive

- SHC_DESTROY

  Note: Destroying the primary stream will suspend the streaming of the secondary streams because all the associated streams use the buffers owned by the primary stream.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_START

  Note: The streaming will only start when the SHC_START command is received for the primary stream, because it owns the I/O thread and the stream buffers.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_DATA_ITEM_NOT_SPECIFIED (stream must be associated before it is started).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_STOP

  Note: A stop for the primary stream will stop all the streams because it owns the I/O thread and the buffers. A stop for a secondary stream will only stop the data for that stream. If a secondary stream is stopped and restarted while the primary stream is still going, the secondary stream data will pick up at the interleaved point where the primary stream is.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_STREAM_NOT_STARTED

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosResumeThread
  - DosSuspendThread

- SHC_SEEK

  Note: SEEK is only valid on the primary stream. A SEEK command on a secondary stream will return ERROR_DATA_ITEM_NOT_SEEKABLE.

  The seek point is granular to the *mmtimePerUnit* specified in the SPCB.

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* or *hid* -or both-passed)
  - ERROR_DATA_ITEM_NOT_SEEKABLE
  - ERROR_DATA_ITEM_NOT_SPECIFIED
  - ERROR_STREAM_NOT_STOP
  - ERROR_SEEK_PAST_END
  - ERROR_SEEK_BEFORE_BEGINNING
  - ERROR_LARGE_SEEK_BY_TIME

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem
  - DosDevIOCtl (ReadLong command to CD device driver).

- SHC_GET_PROTOCOL

  Possible return codes:

- ERROR_INVALID_SPCBKEY

Return codes from the following APIs are also returned:

- DosRequestMutexSem

- SHC_INSTALL_PROTOCOL

  Note: This stream handler allows only data types of DATATYPE_CDXA_AUDIO, DATATYPE_CDXA_VIDEO, or DATATYPE_CDXA_DATA to be installed.

  Possible return codes:

  - ERROR_INVALID_SPCBKEY
  - ERROR_ALLOC_RESOURCES

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_ENUMERATE_PROTOCOLS

  Possible return codes:

  - ERROR_INSUFF_BUFFER

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

- SHC_NEGOTIATE_RESULT

  Possible return codes:

  - ERROR_INVALID_STREAM (invalid *hstream* , *hid*, or both passed)
  - ERROR_INVALID_FUNCTION (must only be called directly after create).

  Return codes from the following APIs are also returned:

  - DosRequestMutexSem

Base Stream Protocol Control Block Data Types Supported

The CD-ROM XA Stream Handler has 7 base SPCBs.

SPCB Field	CDXA_LEVELB Values	CDXA_LEVELC Values
spcbkey.ulDataType	DATATYPE_CDXA_AUDIO	DATATYPE_CDXA_AUDIO
spcbkey.ulDataSubType	CDXA_LEVELB	CDXA_LEVELC
ulDataFlag	SPCBDATA_CUETIME	SPCBDATA_CUETIME
ulNumRec	17	17
ulBlockSize	1	1
ulBufSize	39984	39984
ulMinBuf	8	8
ulMaxBuf	16	16
ulSrcStart	1	1
ulTgtStart	1	1
ulBufFlag	SPCBBUF_INTERLEAVED	SPCBBUF_INTERLEAVED
ulHandFlag	SPCBHAND_NOSYNC | SPCBHAND_PHYS_SEEK	
ulBytesPerUnit	2304	2304

mmtimePerUnit	160	320

SPCB Field	LEVELB_MONO Values	LEVELC_MONO Values
spcbkey.ulDataType	DATATYPE_CDXA_AUDIO	DATATYPE_CDXA_AUDIO
spcbkey.ulDataSubType	CDXA_LEVELB_MONO	CDXA_LEVELC_MONO
ulDataFlag	SPCBDATA_CUETIME	SPCBDATA_CUETIME
ulNumRec	17	17
ulBlockSize	1	1
ulBufSize	39984	39984
ulMinBuf	8	8
ulMaxBuf	16	16
ulSrcStart	1	1
ulTgtStart	1	1
ulBufFlag	SPCBBUF_INTERLEAVED	SPCBBUF_INTERLEAVED
ulHandFlag	SPCBHAND_NOSYNC | SPCBHAND_PHYS_SEEK	
ulBytesPerUnit	2304	2304
mmtimePerUnit	320	640

SPCB Field	CDXA_AUDIO Values
spcbkey.ulDataType	DATATYPE_CDXA_AUDIO
spcbkey.ulDataSubType	CDXA_AUDIO_HD
ulDataFlag	SPCBDATA_CUETIME
ulNumRec	17
ulBlockSize	1
ulBufSize	39984
ulMinBuf	8
ulMaxBuf	16
ulSrcStart	1
ulTgtStart	1
ulBufFlag	SPCBBUF_INTERLEAVED
ulHandFlag	SPCBHAND_NOSYNC | SPCBHAND_PHYS_SEEK
ulBytesPerUnit	0
mmtimePerUnit	0

SPCB Field	CDXA_DATA Values	CDXA_VIDEO Values
spcbkey.ulDataType	DATATYPE_CDXA_DATA	DATATYPE_CDXA_VIDEO
spcbkey.ulDataSubType	0	0
ulNumRec	17	17
ulBlockSize	1	1
ulBufSize	39984	39984
ulMinBuf	8	8
ulMaxBuf	16	16
ulSrcStart	1	1
ulTgtStart	1	1
ulBufFlag	SPCBBUF_INTERLEAVED	SPCBBUF_INTERLEAVED
ulHandFlag	SPCBHAND_NOSYNC | SPCBHAND_PHYS_SEEK	
ulBytesPerUnit	0	0
mmtimePerUnit	0	0

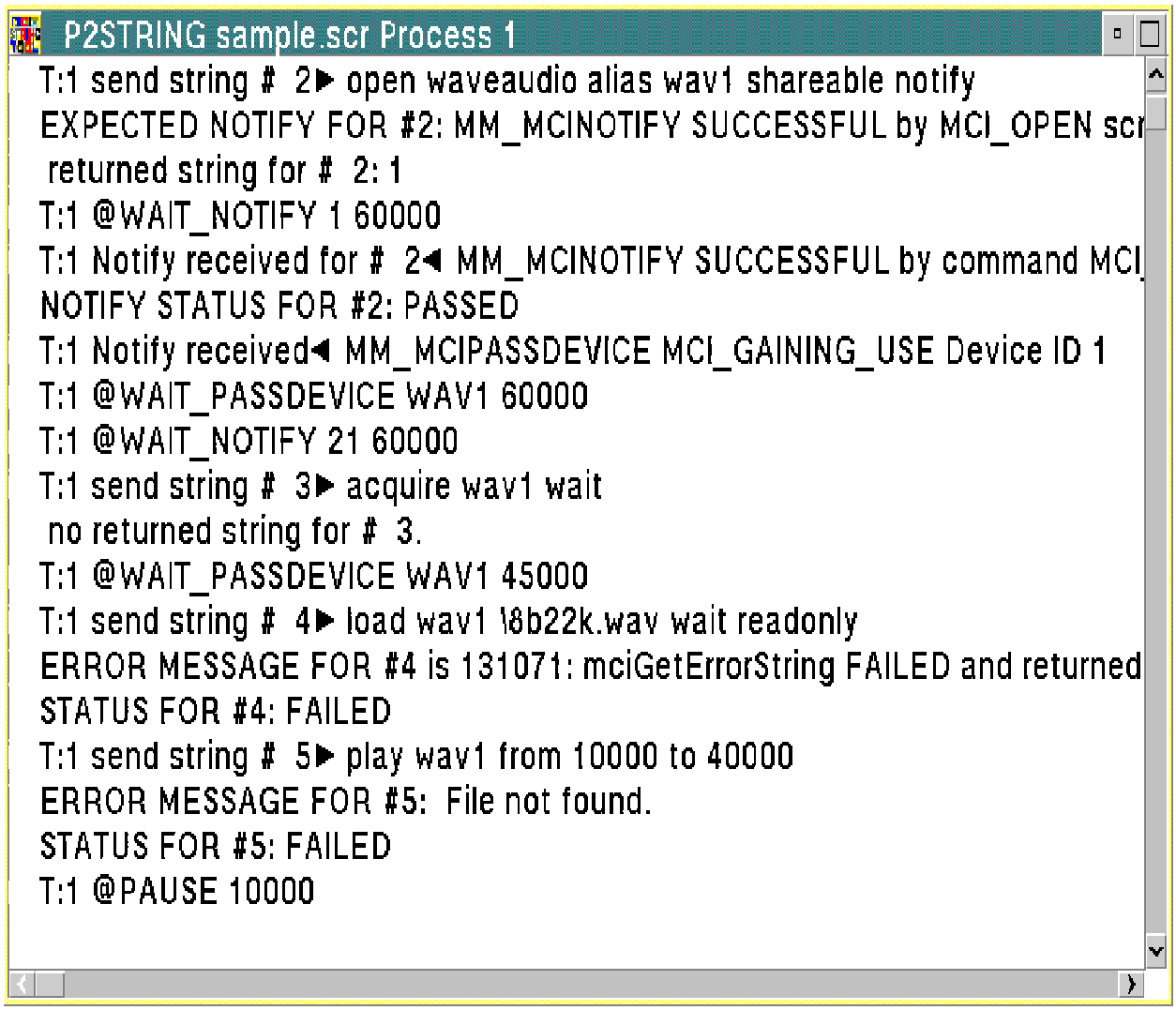
Stream Handler Limits

Maximum number of streams associated to a primary stream is 15.

Maximum number of streams supported by the stream handler at one time is only limited by available memory.

P2STRING Tool

The P2STRING tool processes script files (containing string commands and tool directives) to test the behavior of subsystems in OS/2 multimedia. P2STRING extracts the strings from the script files and processes the commands through the mciSendString function. Messages and error conditions of the processes included in the scripts are logged to an output file and displayed in windows (as shown in the following figure). Each process logs to its own display window, but all processes log to the same output file. In addition, if you close the display window, the string execution thread is immediately destroyed and the entire process ends.



Output messages include all non-comment lines read from the script (for example, script directives, command strings, expected return values, expected and received notify messages, and status lines). In addition, errors and debugging statements of unsuccessful string commands are logged to a file named P2STRING.LOG file.

Setting Font Size and Type

Before you start the P2STRING tool, you can change the size and type of font displayed in the P2STRING window on the desktop. For example, to specify Times Roman font with a 10 point font size, type:

```
SET P2STRING_FONTFACE=TIMES
SET P2STRING_FONTSIZE=10
```

You can specify either font face, font size, or both. Possible FONTFACE values include SYSTEM, COURIER, TIMES, or HELVETICA (default). Possible FONTSIZE values include 8 (default), 10, 12, 14, or 18.

Starting P2STRING

The P2STRING tool consists of two files: P2STRING.EXE and P2S_DLL.DLL. These files are located in the

\TOOLKIT\BIN\BETA\P2STRING subdirectory. The following figure displays the syntax used to start the P2STRING program. Associated parameters are described in the next table in this section.

```
P2STRING inp_file [-a]out_file [-eerr_file] [-d|-D] [-E] [-t]
```

Note: The parameters are case sensitive.

The following table describes the parameters associated with the P2STRING program.

Parameter	Description
script_file	Specifies the script file name you want to process. **Note:** See P2STRING Script Language for information on the contents of a script file and how to interpret the script language.
[-a]*out_file*	Specifies the output file name containing the results of the test. This file contains only the results of the test you are running, unless you specify the optional -a parameter. For example, to append the output of a script file named SAMPLE.SCR to output currently in a file named MDM.OUT, type: `P2STRING SAMPLE.SCR -aMDM.OUT`
[-e*err_file*]	Specifies the optional error file name that receives messages from string commands that completed unsuccessfully. For example, to create an error file named MDM.ERR, type: `P2STRING SAMPLE.SCR MDM.OUT -eMDM.ERR`
[-d|-D]	Specifies one of the following optional parameters: -d Instructs P2STRING to end after processing a script file. Use this parameter when you are running test cases automatically. There is no change in the output. When the script file has completed processing, P2STRING prompts you with a message requiring you to end the test. -D Behaves identically to the -d parameter except that the script directives requiring user input are ignored. **Note:** See Tool Directives for information on how to add execution directives (which require user input) in a script file.
-E	Causes the script file to exit after the first error. By default, script files run to completion regardless of errors. For example, the following command ends the processing of SAMPLE.SCR after an error is encountered: `P2STRING SAMPLE.SCR -aMDM.OUT -d -eMDM.ERR -E`
-t	Records time stamps for strings and MM_MCIPASSDEVICE notification messages.

P2STRING Script Language

This section describes the contents of a script file and how to interpret the script language. Script files can contain the following types of lines:

- Comments
- Tool Directives
- OS/2 Multimedia String Commands
- Expected Return Strings

- Expected Error Messages
- Expected Notification Messages

The following discussion provides information on how to create a script file containing these various line types. The following figure displays an example of how a script file appears.

```
@PROCESSES=2
@EVENTS={HASCTRL1=1,HASCTRL2=0}
#
#
#
@PROCESS 1
;
; set masteraudio level for session to 10% - will affect all
; 3 processes
;
masteraudio volume 10
;
; open waveaudio device non-exclusively
;
open waveaudio alias wav1 shareable notify
+MM_MCINOTIFY MCI_NOTIFY_SUCCESSFUL MCI_OPEN #1
@WAIT_NOTIFY 1 60000
@WAIT_PASSDEVICE wav1 60000
@WAIT_NOTIFY 21 60000
```

Comments

Script comment lines must start with either a *semi-colon* (;) or *pound sign* (#) in the first column. These comment lines are neither displayed nor echoed in the output file. If you want a remark to appear in the output, use the @REM directive.

P2STRING allows for a variable number of lines to be displayed in its window. Regular comment lines (header lines) are not displayed nor written to the output file.

Tool Directives

The P2STRING tool supports either multithreaded or multiprocess execution, but not both. You can use tool directives to test either @PROCESSES or @THREADS in a script file.

Tool directives start with an *at sign* (@) in the first column. These directives affect the execution and appearance of the output. The following classes of directives are recognized:

- Initialization
- Execution

Initialization Directives

Use initialization directives to set up the content of the script file. These directives must appear before execution directives because the tool preprocesses the script file and builds process command buffers.

The @THREADS and @PROCESSES directives are mutually exclusive. In other words, the P2STRING tool supports either multithreaded or multiprocess execution in a script file (not both). In addition, there is a limit of 10 processes or threads per script file. The following table lists the supported initialization directives.

Directive	Description
`@PROCESSES=`*x*	Specifies the number (*x*) of processes the script file will be running. For example: `@PROCESSES=2`
`@THREADS=`*x*	Specifies the number (*x*) of threads the script file will be running.
`@EVENTS={`*n*`[=0|1] [,`*n*`[=0|1]]}`	Specifies one or more *n*ames of events. Events are user-defined with a maximum of 15 characters. Events can be set to 1 or 0. Set an event to 1 for an event that is set with the @SET_EVENT execution directive. Set an event to 0 to clear or reset the event. If no initialization values are specified, the event is initialized to 0. For example: `@EVENTS={brad=1,john=1,test=0}`

Execution Directives

Use execution directives to process the script file. Again, the @THREAD and @PROCESS directives are mutually exclusive.

The following table lists the supported execution directives.

Note: Timing out on the @WAIT_EVENT and @WAIT_NOTIFY directives is *not* considered a failure.

Directive	Description
`@THREAD` *x*	Specifies that the script lines following this directive belong to thread number *x* until the next @THREAD directive is encountered.
`@PROCESS` *x*	Specifies that the script lines following this directive belong to process number *x* until the next @PROCESS directive is encountered.
`@SET_EVENT` *name* `0|1`	Sets the event *name* to either 1 or 0. Use 1 to mark that the event has happened. Use 0 to clear or reset the event. **Note:** The event must be declared through

the @EVENTS directive.

`@WAIT_EVENT` *name* [*to*]

Waits until the event *name* is set to 1. @WAIT_EVENT does not cause a change of the event state. If you need a reusable event, use this directive.

The timeout (*to*) is specified in milliseconds. If omitted, it defaults to 3 minutes.

`@WAIT_NOTIFY` *x* [*to*]

Waits for a specific number (*x*) from an expected MM_MCINOTIFY notification message. This number must match the index used in the MM_MCINOTIFY reference line. The @WAIT_NOTIFY events are not reusable because there are no events that logically reset it.

The timeout (*to*) is specified in milliseconds. If omitted, it defaults to 3 minutes.

If the associated mciSendString function fails, the event is posted to prevent delays for notifications that are never going to be sent.

`@WAIT_PASSDEVICE` *alias* [*to*]

Waits until the device instance with an alias of *alias* gains use. This assumes that the alias names used within a script file are unique. The maximum alias name length is 20 characters.

Note: Use a unique alias for every OPEN command.

The timeout (*to*) is specified in milliseconds. If omitted, it defaults to 3 minutes.

The tool assumes that a unique alias is specified on each OPEN string command. If unique aliases are not used, errors conditions might occur.

`@REM` *comment*

Echoes the *comment* to the screen and the output log file. All other script comment lines (those starting with **;** or **#**) are neither transferred nor displayed.

`@PAUSE` *to*

Pauses processing of the current thread or process in the input script file for the specified time. It does not stop the processing of the notifications or window functions. Other threads or processes are not affected by this directive.

`@BREAK` [*message*]

Causes a message box to appear with

message text. Script processing is halted until the user responds with the correct action. For example:

```
@BREAK [replace the CD]
```

`@CHECK [`*`message`*`]`

Grades the success of the previous command based on the user's response. A pop-up window displays the *message* and prompts the user with Yes or No push buttons. The status is passed if the user selects Yes, or failed if No is selected. For example:

```
@BREAK The music will play for 5 secs.  R
play cdaudio notify
@PAUSE 5000
@CHECK Did it play?
```

OS/2 Multimedia String Commands

All lines that do not fall into any of the other categories are interpreted as OS/2 multimedia string commands. These lines are passed to the Media Device Manager (MDM) through the mciSendString function after the environment variables have been substituted.

Any token in the string command line bracketed by question marks (such as ?FOO?) is interpreted as an environment variable. The actual value of the environment variable is substituted into the string before it is passed to the mciSendString function. If the variable is not found, a warning is issued and the token is replaced with a NULL string. For example, assuming the environment string MMDATA is set to D:\DATA, **open ?mmdata?\temp.wav alias a** is equal to **open d:\data\temp.wav alias a**.

Expected Return Strings

Many OS/2 multimedia commands return strings. It is possible to check these strings against an expected value with an expected return string line.

An expected return string line has the format:

```
=result
```

The *equal sign* (=) must be in column 1 and should have no trailing spaces. If an empty string is expected, nothing should follow the = (not even spaces). For example:

```
status cdaudio ready wait
=TRUE
status cdaudio mode wait
=stopped
```

The expected result is always interpreted as a string. This might produce some unusual outputs for commands that return binary data.

Where the return is a textual numerical value, it may be matched to a tolerance range of ±10% using a *tilde* (~) before the number. This is typically used when the exact value cannot be known. For example:

```
set foo time format milliseconds wait
```

```
play foo notify
@PAUSE 1000
stop foo wait
status foo position wait
=~1000
```

Thus, the status command is considered successful if the returned string is in the range 900 - 1100.

Expected Error Messages

When an OS/2 multimedia string command is expected to fail with an error, use the expected error line to specify the expected error. The expected error line has the format:

```
=!error
```

The =! must start in column 1. If any error is acceptable, then use the keyword ERROR. If a specific error is expected, enter the exact error message after the =! . For example:

```
open sequencer alias mymidi wait
load mymidi nofile.foo
=!File not found.
```

Be careful about extra blanks in the expected-error and expected-result lines. The case of the strings is unimportant; however the comparison will fail if the spacing or punctuation does not match exactly.

Expected Notification Messages

Many OS/2 multimedia string commands cause notification messages to be sent to the P2STRING tool. The system uses notification messages to respond to applications. For example, notification messages indicate system status regarding completion of a media device function or passing of the ownership of a media control device from one process to another.

It is possible to verify that the proper notifications are received by using an expected-notify line. Each expected notification line begins with a *plus sign* (+) in column 1. The following types of notification lines are possible:

- Command completion/error notifications
- Event notifications
- Position change notifications

Note: Any or all notification messages might be expected for a single OS/2 multimedia string command.

Command Completion/Error Notifications

An MM_MCINOTIFY line notifies an application when a device successfully completes the action indicated by a media message or when an error occurs.

```
+MM_MCINOTIFY  notify-code[#x]
```

Keyword	Description
notify-code	Specifies the notification message code, for example, MCI_NOTIFY_SUCCESSFUL. The spelling must be the same as the #defines in the OS2ME.H file for the notify codes.
message	Specifies the media control interface message that caused the notification, for example, MCI_PLAY. The spelling must be the same as the #defines in the OS2ME.H file for MCI messages.
x	Specifies a unique number (*x*) used to associate @WAIT_NOTIFY directives with particular notifications. This number has nothing to do with the order in which strings are sent. The number must be unique and in the range 1-99.

Event Notifications

An MM_MCIPOSITIONCHANGE line notifies an application of the current media position.

+MM_MCIPOSITIONCHANGE *position %user-parameter #x*

Keyword	Description
position	Specifies the expected MMTIME position of the first position-change message.
user-parameter	Specifies the user parameter to be returned. This should be the same as the return *value* specified in the SETPOSITIONADVISE string command. Note that the return value must be a unique integer in the range of 1-99.
x	Specifies the number (*x*) of position change messages expected. For further information, see Limitations of MM_MCIPOSITIONCHANGE Verification.

Position Change Notifications

An MM_MCICUEPOINT line notifies an application that the playlist processor has encountered a *message* instruction.

+MM_MCICUEPOINT *position %user-parameter*

Keyword	Description
position	Specifies the expected MMTIME position of the first position-change message.

user-parameter	Specifies the user parameter to be returned. This should be the same as the return *value* specified in the SETPOSITIONADVISE string command. Note that the return value must be a unique integer in the range of 1-99.

Note: Other notifications cannot be compared because they do not allow for a *user-parameter* as part of the message, which is essential for the tracking of related notifications.

Limitations of MM_MCIPOSITIONCHANGE Verification

There are limitation to what you can verify in the P2STRING tool using event notification lines. The MM_MCIPOSITIONCHANGE line requires the use of the "return *value* " item in the respective string commands. This line also does not provide for timing start point; for example, playing has started. The P2STRING tool can only count the number of messages received for a specific user parameter (used as a key) and check if subsequent messages have positions approximate to the given expected position interval. The script writer must determine how many SETPOSITIONADVISE messages are expected, considering the duration of playing, time format, and start position of the play/record. The reference *position* given in the expected notification line must be in MMTIME units. If the "expected number of messages" parameter is omitted, the tool only verifies the position interval (not the number). In case of scripts where play, seek, or record are used to cover non-monotonic ranges, P2STRING might report failures on position-advises because it expects each SETPOSITIONADVISE to be in the next position interval from the previous message. If the script makes a jump to a position that does not conform to this, the status will be logged as failed.

For example:

```
setpositionadvise SomeDevice every 10000 on return 5
+MM_MCIPOSITIONCHANGE 10000 %5
play SomeDevice from 35000 to 55000 notify (produce 3 positionchange msgs)
seek SomeDevice to 30000 wait
play SomeDevice notify (produces a number of messages starting at 30000)
```

MM_MCIPOSITIONCHANGE messages are logged as failed, because of the lapse in position interval. A way to handle this situation is to disable the MM_MCIPOSITIONCHANGE before an explicit position jump is made and enable the same SETPOSITIONADVISE with a different user parameter.

For example:

```
setpositionadvise SomeDevice every 10000 on return 5
+MM_MCIPOSITIONCHANGE 10000 %5
play SomeDevice from 35000 to 55000 notify (produce 3 positionchange msgs)
setpositionadvise SomeDevice every 10000 off
seek SomeDevice to 30000 wait
setpositionadvise SomeDevice every 10000 on return 6
+MM_MCIPOSITIONCHANGE 10000 %6
play SomeDevice notify (produce a number of messages starting at 30000)
```

Processing Logic

A string command line can be followed by zero to one return value lines, or zero to three notification lines. Note that this is an exclusive OR, meaning that specifying both expected return value and expected notification is not going to give reliable results due to the fact that the returned buffer does not become valid prior to the end of the notify message. Also, in case of notify flags, return values are in the procedural interface format rather than in the string interface format.

The MDM will not be able to convert return values to strings for commands processed with the notify flag because media control drivers will be sending their notify messages directly to the application.

Status of each command is determined in two stages. The first stage is at string execution. If the mciSendString function returns an error and there was no =!ERROR reference line in the script following the command string line, the command is considered failed. If a return value was found after mciSendString is processed, the tool will check for expected return and perform comparison of the two. It they do not match, the command is considered failed. In case of an error that is not in the range understood by the mciGetErrorString function, the command is considered failed even if the !ERROR was encountered.

The second stage of the comparison is after a notification is received and after all the commands are issued. If a notification was received and it successfully compared to the expected notification line, the command is considered successful. If there was no reference notification line, status of the command will not be assigned, unless notify-error was returned. After all the scripts are processed, expected reference notifications will be used to determine if all the notifications were received. The commands that did not receive a notify, and had an expected notification line of the type, are marked failed. Note that command strings are not examined for presence of a notify flag and it is the writer's responsibility to create an expected notify line if it is of importance. In case of expected NOTIFY_SUCCESSFUL messages, all codes other than NOTIFY_ERROR are considered valid. This includes NOTIFY_SUCCESSFUL, NOTIFY_ABORTED and NOTIFY_SUPERCEDED. If any other notify code was specified as expected, and exact match will be checked for. If NOTIFY_ERROR is expected, all errors in the range are verified as passed. There is no facility for verification of an exact error code returned in the notification.

Notices

August 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time

This publication was produced in the United States of America. IBM may not offer the products, services, or features discussed in this document in other countries, and the information is subject to change without notice. Consult your local IBM representative for information on the products, services, and features available in your area.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in

conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Audio Visual Connection
Common User Access
CUA
IBM
Multimedia Presentation Manager/2
OS/2
Presentation Manager
Workplace Shell

The following terms are trademarks of other companies:

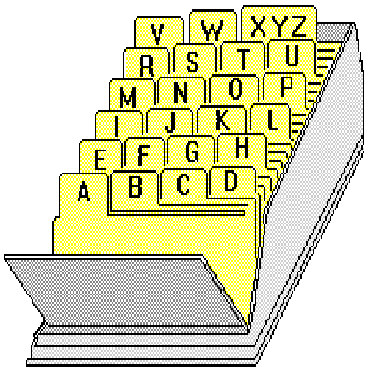
Helvetica
Pro AudioSpectrum 16
Sound Blaster
Linotype Company
Media Vision, Inc.
Creative Technology Ltd.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

Glossary

Select the starting letter of the glossary term you want to locate.



A

AB roll - Synchronized playback of two recorded video images so that one can perform effects, such as dissolves, wipes, or inserts, using both images simultaneously.

ACPA - Audio capture and playback adapter.

active matrix - A technology that gives every pel (dot) on the screen its own transistor to control it more accurately. (This allows for better contrast and less motion smearing.)

adaptive differential pulse code modulation - A bit-rate reduction technique where the difference in pulse code modulation samples are not compressed before being stored.

ADC - Analog-to-digital converter.

ADPCM - Adaptive differential pulse code modulation.

aliasing - The phenomenon of generating a false (alias) frequency, along with the correct one, as an artifact of sampling a signal at discrete points. In audio, this produces a "buzz." In imagery, this produces a jagged edge, or stair-step effect. See also *moire* .

all points addressable (APA) - In computer graphics, pertaining to the ability to address and display or not display each picture element on a display surface.

AM - Animation metafile.

ambience - In audio, the reverberation pattern of a particular concert hall, or listening space.

ambient noise - In acoustics, the noise associated with a particular environment, usually a composite of sounds from many distant or nearby sources.

American National Standards Institute (ANSI) - An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

AMF - Animation metafile format.

amp - See *amplifier* .

amplifier - (1) A device that increases the strength of input signals (either voltage or current). (2) Also referred to as an *amp* .

amp-mixer - (1) A combination amplifier and mixer that is used to control the characteristics of an audio signal from one or more audio sources. (2) Also referred to as an amplifier-mixer.

analog - Pertaining to data consisting of continuously variable physical quantities. Contrast with *digital* .

analog audio - Audio in which all information representing sounds is stored or transmitted in a continuous-scale electrical signal, such as line level audio in stereo components. See also *digital audio.*

analog-to-digital converter (ADC) - A functional unit that converts data from an analog representation to a digital representation. (I) (A)

analog video - Video in which all the information representing images is in a continuous-scale electrical signal for both amplitude and time. See also *digital video.*

analog video overlay - See *overlay* .

anchor - An area of a display screen that is activated to accept user input. Synonymous with *hot spot, touch area,* and *trigger* .

animate - Make or design in such a way as to create apparently spontaneous, lifelike movement.

animated - Having the appearance of something alive.

animated screen capture - Recording a computing session for replay on a similar computer with voice annotation. (An example is sending a spreadsheet with an accompanying screen recording as an explanation and overview.)

animatic - A limited animation consisting of artwork shot on film or videotape and edited to serve as an on-screen storyboard.

animation metafile - A compound file format, the elements of which are the frames of animation themselves. These frames are stored sequentially so that they can be played back in time by streaming to the video agent.

animation metafile format (AMF) - The file format used to store animated frame sequences.

annotation - The linking of an object with another, where the second contains some information related to the first. For example, an audio annotation of a spreadsheet cell might contain verbal explanation about the contents of the cell.

ANSI - American National Standards Institute.

anthropomorphic software agent - The concept of a simulated agent, seemingly living inside the computer, that talks to and listens to the user, and then acts for the user on command.

antialiasing - (1) In imagery, using several intensities of colors (a ramp) between the color of the line and the background color to create the effect of smoother curves and fewer jagged edges on curves and diagonals. (2) In imagery or audio, removing aliases by eliminating frequencies above half the sample frequencies.

AOCA - Audio Object Content Architecture.

APA - All points addressable.

APA graphics - All Points Addressable graphics. See *bitmap graphics* .

API - Application programming interface.

application programming interface (API) - A functional interface supplied by the operating system or an IBM separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application-supplied video window - (1) An application can specify to MMPM/2 that it wants video played in a specific window (controlled by the application) instead of the default window (controlled by MMPM/2). The application supplied video window can be used to implement advanced features not supported by the default video window. (2) Also referred to as an *alternate video window* .

artifact - A product resulting from human activity; in computer activity, a (usually unwanted) by-product of a process.

aspect ratio - (1) On a display screen, the ratio of the maximum length of a display line to the maximum length of a display column. (2) The ratio of height to width. (This term applies to areas or individual pels.) Refer to *enhanced graphics adapter* and *video graphics adapter* .

asymmetric video compression - In multimedia applications, the use of a powerful computer to compress a video for mastering so that a less powerful (less expensive) system is needed to decompress it. Contrast with *symmetric video compression* .

audible cue - A sound generated by the computer to draw a user's attention to, or provide feedback about, an event or state of the computer. Audible cues enhance and reinforce visible cues.

audio - Pertaining to the portion of recorded information that can be heard.

audio attribute control - Provides access to and operation of the standard audio attributes: mute, volume, balance, treble, and bass. All device communication and user interface support is handled by the control.

audio attributes - Refers to the standard audio attributes: mute, volume, balance, treble and bass.

audio clip - A section of recorded audio material.

Audio Object Content Architecture - A data format for multimedia products.

audio processing - Manipulating digital audio to, for example, edit or create special effects.

audio segment - A contiguous set of recorded data from an audio track. An audio segment might or might not be associated with a video segment.

audio track - (1) The audio (sound) portion of the program. (2) The physical location where the audio is placed beside the image. (A system with two audio tracks can have either stereo sound or two independent audio tracks.) (3) Synonym for *sound track* .

audiovisual - A generic term referring to experiences, equipment, and materials used for communication that make use of both hearing and sight.

Audio Visual Connection* (AVC) - An authoring system used on an IBM PS/2* to develop and display audiovisual productions.

audiovisual computer program - A computer program that makes use of both hearing and sight.

authoring - A structured approach to combining all media elements within an interactive production, assisted by computer software designed for this purpose.

authoring system - A set of tools used to create an interactive multimedia application without implementing formal programming.

AVI file format - The Audio/Video Interleaved (AVI) file format is the standard file format used to support software motion video. AVI files can contain multiple streams (tracks) of data (for example, a video and an audio stream). The streams are interleaved to improve access times during playback. The present implementation is limited to a single video stream and a single, optional, audio stream.

B

background image - The part of a display image, such as a form overlay, that is not changed during a particular sequence of transactions. Contrast with *foreground image* .

background music - In videotaping, music that accompanies dialog or action.

balance - For audio, refers to the relative strength of the left and right channels. A balance level of 0 is left channel only. A balance level of 100 is right channel only.

basic input/output system (BIOS) - In an IBM personal computer, microcode that controls basic hardware operations, such as interactions with diskette drives, hard disk drives, and the keyboard. (For example, the IBM Enhanced Graphics Adapter has an addressable BIOS, located on the adapter itself, that is used to control the IBM InfoWindow* graphics functions.)

beta format - A consumer and industrial 0.5-inch tape format.

BG - Script abbreviation for *background* .

BIOS - Basic input/output system.

bitmap - A coded representation in which each bit, or group of bits, represents or corresponds to an item, for example, a configuration of bits in main storage in which each bit indicates whether a peripheral device or a storage block is available or in which each group of bits corresponds to one pel of a display image.

bit-block transfer - Transfer of a rectangular array of bitmap data.

bitblt - Bit-block transfer. Synonym for *blit* .

bitmap graphics - A form of graphics whereby all points on the display are directly addressable. See also *all points addressable* .

blit - Synonym for *bitblt* .

blitter - Hardware that performs bit-block transfer operations.

BND - (1) An internal I/O procedure, provided by the MMPM/2 system, that supports RIFF compound files (commonly called *bundle files*). (2) The four-character code (*FOURCC*) of a bundle file. (3) See also *RIFF compound file* .

BND file - A RIFF compound file.

BND IOProc - An internal I/O procedure, provided by the MMPM/2 system, that supports the elements in a RIFF compound file. See also *CF IOProc* .

boom - A long, relatively horizontal supporting brace used for holding a microphone or camera; sometimes used to refer to the machinery

that supports the camera and allows it to move while shooting.

brightness - Refers to the level of luminosity of the video signal. A brightness level of 0 produces a maximally white signal. A brightness level of 100 produces a maximally black signal.

buffer - A portion of storage used to hold input or output data temporarily.

bundle file (BND) - (1) A file that contains many individual files, called *file elements* , bound together. The MMIO file manager provides services to locate, query, and access file elements in a bundle file. (2) A RIFF compound file.

bus - A facility for transferring data between several devices located between two end points, only one device being able to transmit at a given moment.

buy - (1) In videotaping, footage that is judged acceptable for use in the final video. (2) Synonym for *keeper* .

C

CAI - Computer-assisted instruction. Synonym for *CBT* .

calibration - The adjustment of a piece of equipment so that it meets normal operational standards. (For example, for the IBM InfoWindow system, calibration refers to touching a series of points on the screen so that the system can accurately determine their location for further reference.)

camcorder - A compact, hand-held video camera with integrated videotape recorder.

capture - To take a snapshot of motion video and retain it in memory. The video image may then be saved to a file or restored to the display.

cast animation - (1) A sequence of frames consisting of manipulations of graphical objects. (2) The action of bringing a computer program, routine, or subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (3) See also *frame animation* .

CAV - Constant angular velocity.

CBT - Computer-based training. Synonym for *CAI* .

CCITT - Comite Consultatif International Telegraphique et Telephonique. The International Telegraph and Telephone Consultative Committee.

CD - Compact disc.

CD-DA - Compact disc, digital audio.

CD-I - Compact Disc-Interactive.

CD-ROM - Compact disc, read-only memory.

CD-ROM XA - Compact disc, read-only memory extended architecture.

cel - A single frame (display screen) of an animation. (The term originated in cartooning days when the artist drew each image on a sheet of celluloid film.)

CF IOProc - An internal I/O procedure, provided by the MMPM/2 system, that supports RIFF compound files. The CF IOProc operates on the entire compound file (rather than on the elements in a RIFF compound file, as with the BND IOProc). The CF IOProc is limited to operations required by the system to ensure storage system transparency at the application level. See also *BND IOProc* .

CGA - Color graphics adapter.

CGRP - Compound file resource group.

channel mapping - The translation of a MIDI channel number for a sending device to an appropriate channel for a receiving device.

channel message - A type of non-SysEx MIDI message that has a channel identifier in it, implying that these messages are specific to one channel.

check disc - A videodisc produced from the glass master that is used to check the quality of the finished *interactive program* .

chord - To press more than one button at a time on a pointing device.

chroma-key color - The specified first color in a combined signal. See also *chroma-keying* .

chroma-keying - Combining two video signals that are in sync. The combined signal is the second signal whenever the first is of some specified color, called the chroma-key color, and is the first signal otherwise. (For example, the weatherman stands in front of a blue background-blue is the chroma-key color.) At home, the TV viewer sees the weather map in place of the chroma-key color, with the weatherman suspended in front.

chroma signal - The portion of image information that provides the color (hue and saturation).

chrominance - The difference between a color and a reference white of the same luminous intensity.

chunk - (1) The basic building block of a RIFF file. (2) A RIFF term for a formalized data area. There are different types of chunks, depending on the chunk ID. (3) See *LIST chunk* and *RIFF chunk* .

chunk ID - A four-character code (FOURCC) that identifies the representation of the chunk data.

circular slider control - A knob-like control that performs like a control on a TV or stereo.

circular slider knob - A knob-like dial that operates like a control on a television or stereo.

class - (1) A categorization or grouping of objects that share similar behaviors and characteristics. Synonym for *object class.* (2) See *node class* . (RTMIDI-specific term)

click - To press and release a button on a pointing device without moving the pointer off the choice. See *double-click* . See also *drag select* .

clip - A section of recorded, filmed, or videotaped material. See also *audio clip* and *video clip* .

closed circuit - A system of transmitting television signals from a point of origin to one or many restricted destination points specially equipped to receive the signals.

close-up - In videotaping, the picture obtained when the camera is positioned to show only the head and shoulders of a subject; in the case of an object, the camera is close enough to show details clearly. See also *extreme close-up* .

CLP - Common loader primitive.

CLUT - Color look-up table. Synonym for *color palette* .

CLV - Constant linear velocity.

CODEC - compressor/decompressor (CODEC) - An algorithm implemented either in hardware or software that can either compress or decompress a data object. For example, a CODEC can compress raw digital images into a smaller form so that they use less storage space. When used in the context of playing motion video, decompressors reconstruct the original image from the compressed data. This is done at a high rate of speed to simulate motion.

collaborative document production - A system feature that provides the ability for a group of people to manage document production.

collision - An unwanted condition that results from concurrent transmissions on a channel. (T) (For example, an overlapping condition that occurs when one sprite hides another as it passes over it.)

color cycling - An animation effect in which colors in a series are displayed in rapid succession.

color graphics adapter (CGA) - An adapter that simultaneously provides four colors and is supported on all IBM Personal Computer and Personal System/2* models.

colorization - The color tinting of a monochrome original.

color palette - (1) A set of colors that can be displayed on the screen at one time. This can be a standard set used for all images or a set that can be customized for each image. (2) Synonym for *CLUT* . (3) See also *standard palette* and *custom palette* .

common loader primitive - A system service that provides a high-level interface to hardware-specific loaders.

common user access (CUA) - (1) Guidelines for the dialog between a human and a workstation or terminal. (2) One of the three SAA architectural areas.

compact disc (CD) - A disc, usually 4.75 inches in diameter, from which data is read optically by means of a laser.

compact disc, digital audio (CD-DA) - The specification for audio compact discs. See also *Redbook audio* .

Compact Disc-Interactive (CD-I) - A low-cost computer, being developed by N.V. Phillips (The Netherlands) and Sony (Japan), that plugs into standard television sets to display text and video stored on compact discs.

compact disc, read-only memory (CD-ROM) - High-capacity, read-only memory in the form of an optically read compact disc.

compact disc, read-only memory extended architecture (CD-ROM XA) - An extension to CD-ROM supporting additional audio and video levels for compression and interlacing of audio, video, and digital data.

component video - A video signal using three signals, one of which is luminance, and the other two of which are the color vectors. See also *composite video* and *S-video* .

composed view - A view of an object in which relationships of the parts contribute to the overall meaning. Composed views are provided primarily for data objects.

composite - The combination of two or more film, video, or electronic images into a single frame or display. See also *composite video* .

composite monitor - A monitor that can decode a color image from a single signal, such as NTSC or PAL. Contrast with *RGB* .

composite object - An object that contains other objects. For example, a document object that contains not only text, but graphics, audio, image, and/or video objects, each of which can be manipulated separately as an individual object.

composite video - A single signal composed of chroma, luminance, and sync. NTSC is the composite video that is currently the U.S. standard for television. See also *component video* and *S-video* .

compound device - A multimedia device model for hardware that requires additional data objects, referred to as data elements, before multimedia operations can be performed.

compound file - A file that contains multiple file elements.

compound file resource group (CGRP) - A RIFF chunk that contains all the compound file elements, concatenated together.

compound file table of contents (CTOC) - A *RIFF chunk* that indexes the CGRP chunk, which contains the actual multimedia data elements. Each entry contains the name of, and other information about, the element, including the offset of the element within the CGRP chunk. All the *CTOC* entries of a table are of the same length and can be specified when the file is created.

compound message - A structure which combines a time stamp, a source instance identifier, a track number, and a MIDI message. Each of these fields is 32 bits, so the structure is 16 bytes in length. (RTMIDI-specific term)

computer-animated graphics - Graphics animated by using a computer, compared to using videotape or film.

computer-assisted instruction (CAI) - (1) A data processing application in which a computing system is used to assist in the instruction of students. The application usually involves a dialog between the student and a computer program. An example is the OS/2 tutorial. (2) Synonym for *computer-based training* .

computer-based training (CBT) - Synonym for *computer-assisted instruction* .

computer-controlled device - An external video source device with frame-stepping capability, usually a videodisc player, whose output can be controlled by the multimedia subsystem.

conforming - Performing final editing on film or video using an offline edited master as a guide.

connection - The establishment of the flow of information from a connector on one device to a compatible connector on another device. A connection can be made that is dependent on a physical connection, for example the attachment of a speaker to an audio adapter with a speaker wire. A connection can also be made that is completely internal to the PC, such as the connection between the waveaudio media device and the ampmix device. See also *connector* .

connector - A software representation of the physical way in which multimedia data moves from one device to another. A connector can have an external representation, such as a headphone jack on a CD-ROM player. A connector can also have an internal representation, such as the flow of digital information into an audio adapter. See also *connection* .

constant angular velocity (CAV) - Refers to both the format of data stored on a videodisc and the videodisc player rotational characteristics. CAV videodiscs contain 1 frame per track. This allows approximately 30 minutes of playing time per videodisc side. CAV videodisc players spin at a constant rotational speed (1800 rpm for NTSC or 1500 rpm for PAL) and play 1 frame per disc revolution. CAV players support *frame-accurate searches* . See also *constant linear velocity* .

constant linear velocity (CLV) - Refers to both the format of data stored on a videodisc and the videodisc player characteristics. CLV videodiscs contain 1 frame on the innermost track and 3 frames of data on the outermost track. This allows approximately 1 hour of playing time per videodisc side. CLV videodisc players vary the rotational speed from approximately 1800 rpm at the inner tracks to 600 rpm at the outer tracks (for NTSC).

Currently, few CLV players support *frame-accurate searches* They only support search or play to within one second (30 frames for NTSC or 25 frames for PAL). See also *constant angular velocity* .

container - An object whose specific purpose is to hold other objects. A folder is an example of a container object.

contents view - A view of an object that shows the contents of the object in list form. Contents views are provided for container objects and for any object that has container behavior, for example, a device object such as a printer.

continuity - In videotaping, consistency maintained from shot to shot and throughout the take. For example, a switch that is on in one shot should not be off in the next unless it was shown being turned off.

continuous media object - A data object that varies over time; a stream-oriented data object. Examples include audio, animation, and video.

contrast - The difference in brightness or color between a display image and the area in which it is displayed. A contrast level of 0 is minimum difference. A contrast level of 100 is maximum difference.

control - A visual user interface component that allows a user to interact with data.

coordinate graphics - (1) Computer graphics in which display images are generated from display commands and coordinate data. (2) Contrast with *raster graphics* . (3) Synonym for *line graphics* .

crop - To cut off; to trim (for example, a tape).

crossfade - Synonym for *dissolve* .

cross-platform - Used to describe applications that are operable with more than one operating system.

cross-platform transmission - Electronic transmission of information (such as mail) between incompatible operating systems.

CTOC - Compound file table of contents.

CU - Script abbreviation for *close-up* .

CUA - Common User Access.

cue point - A point that the system recognizes as a signal that may be acted upon.

custom palette - (1) A set of colors that is unique to one image or one application. (2) See also *standard palette* and *color palette* .

cut - The procedure of instantly replacing a picture from one source with a picture from another. (This is the most common form of editing scene to scene.)

D

DAC - Digital-to-analog converter.

data object - In an application, an element of a data structure (such as a file, an array, or an operand) that is needed for program execution and that is named or otherwise specified by the allowable character set of the language in which the program is coded.

data stream - All data transmitted through a data channel.

data streaming - Real-time, continuous flowing of data.

DCP - See *device control panel* .

decode - (1) To convert data by reversing the effect of previous encoding. (2) To interpret a code. (3) To convert encoded text into plaintext by means of a code system. (4) Contrast with *encode* .

default video window - (1) Refers to where video is displayed when an application does not indicate an application-defined window with the MCI_WINDOW message. This is provided by and managed for the application by MMPM/2. (2) See also *application-defined window* .

default window - See *default video window* .

delta frame - Refers to one or more frames occurring between reference frames in the output stream. Unlike a reference frame, which stores a complete image, a delta frame stores only the changes in the image from one frame to the next. See *reference frame* .

destination rectangle - An abstract region which defines the size of an image to be created when recording images for software motion video playback. The ratio of this rectangle's size to that of the source rectangle determines the scaling factor to be applied to the video.

destination window - See *destination rectangle* .

device capabilities - The functionality of a device, including supported component functions.

device context - The device status and characteristics associated with an opened instance of an Media Control Interface device.

device control panel (DCP) - An integrated set of controls that is used to control a device or media object (such as by playing, rewinding, increasing volume, and so on).

device controls - See *standard multimedia device controls* .

device driver - (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, a videodisc player, or a CD drive.

device element - A data object, such as a file, utilized by a compound device.

device object - An object that provides a means for communication between a computer and the outside world. A printer is an example of a device object.

device sharing - (1) The ability to share a device among many different applications simultaneously. If a device is opened shareable, the device context will be saved by the operating system when going from one application to another application. (2) Allowing a device context to be switched between Media Control Interface devices.

device-specific format - The storage or transmission format used by a device, especially if it is different from an accepted standard.

dialog - In an interactive system, a series of related inquiries and responses similar to a conversation between two people.

digital - (1) Pertaining to data in the form of numeric characters. (2) Contrast with *analog* .

digital audio - (1) Material that can be heard that has been converted to digital form. (2) Synonym for *digitized audio* .

digital signal processor (DSP) - A high-speed coprocessor designed to do real-time manipulation of signals.

digital video - (1) Material that can be seen that has been converted to digital form. (2) Synonym for *digitized video* .

digital video effects (DVE) - An online editing technique that manipulates on-screen a full video image; activity for creating sophisticated transitions and special effects. Digital video effects (DVE) can involve moving, enlarging, or overlaying pictures.

Digital Video Interactive (DVI) - A system for bringing full-screen, full-motion television pictures and sound to a regular PC. DVI is a chip set and uses delta compression; that is, only the image-to-image changes in each frame are saved rather than the whole frame. Data (video footage) is compressed into a form that reduces memory requirements by factors of 100 or greater. This compressed data is stored on optical discs and can be retrieved at a rate of 30 frames per second. (The DVI technology was developed by RCA and then sold to Intel. IBM has chosen this technology for future use in the PS/2.)

digital-to-analog converter (DAC) - (1) A functional unit that converts data from a digital representation to an analog representation. (2) A device that converts a digital value to a proportional analog signal.

digitize - To convert an analog signal into digital format. (An analog signal during conversion must be *sampled* at discrete points and quantized to discrete numbers.)

digitized audio - Synonym for *digital audio* .

digitized video - Synonym for *digital video* .

digitizer - A device that converts to digital format any image captured by the camera.

direct manipulation - A set of techniques that allow a user to work with an object by dragging it with a pointing device or interacting with its pop-up menu.

direct memory access - The transfer of data between memory and input and output units without processor intervention.

direct-read-after-write (DRAW) disc - A videodisc produced directly from a videotape, one copy at a time. A DRAW disc usually is used to check program material and author applications before replicated discs are available.

disc - Alternate spelling for *disk* .

discard stop - In data streaming, requests that the data stream be stopped and the data remaining in the stream buffers be discarded.

disk - A round, flat, data medium that is rotated in order to read or write data.

display image - A collection of display elements or segments that are represented together at any one time on a display surface. See also *background image* and *foreground image* .

dissolve - To fade down one picture as the next fades up. Synonym for *crossfade* .

dithering - When different pixels in an image are prebiased with a varying threshold to produce a more continuous gray scale despite a limited palette. This technique is used to soften a color line or shape. This technique also is used for alternating pixel colors to create the illusion of a third color.

DLL - Dynamic-link library.

dolly - A wheeled platform for a camera; a camera movement where the tripod on which the camera is mounted physically moves toward or away from the subject.

DOS IOProc - An internal I/O procedure, provided by the MMPM/2 system, that supports DOS files.

double-click - In SAA Advanced Common User Access, to press and release a mouse button twice within a time frame defined by the user, without moving the pointer off the choice. See *click* . See also *drag select* .

drag select - In SAA Advanced Common User Access, to press a mouse button and hold it down while moving the pointer so that the pointer travels to a different location on the screen. Dragging ends when the mouse button is released. All items between the button-down and button-up points are selected. See also *click* , *double-click* .

DRAW disc - Direct-read-after-write disc.

drop-frame time code - A nonsequential time code used to keep tape time code matched to real time. Must not be used in tapes intended for videodisc mastering.

DSP - Digital signal processor.

DTMF - Dual-tone modulation frequency.

dual plane video system - Refers to when graphics from the graphics adapter are separate from the analog video. That is, there is a separate graphics plane and video plane. The analog video appears behind the graphics, showing through only in the areas that are transparent. Since graphics and video are separate, capturing the graphics screen will only obtain graphics, and capturing the video screen will only obtain video. This is also true for restoring images. See also *single plane video system* .

dual-state push button - A push button that has two states, in and out. It is used for setting and resetting complementary states, such as Mute and Unmute.

dual-tone modulation frequency (DTMF) - Pushbutton phone tones.

dub - To copy a tape; to add (sound effects or new dialog) to a film; to provide a new audio track of dialog in a different language. (Often used with "in" as "dub in".)

DVE - Digital video effects.

DVI - Digital Video Interface

dynamic icon - An icon that changes to convey some information about the object that it represents. For example, a folder icon can show a count, indicating the number of objects contained within the folder. Also, a tape player icon can show an animation of turning wheels to indicate that the machine playing.

dynamic linking - In the OS/2 operating system, the delayed connection of a program to a routine until load time or run time.

dynamic link library (DLL) - A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic resource allocation - An allocation technique in which the resources assigned for execution of computer programs are determined by criteria applied at the moment of need. (I) (A)

dynamic resource - A multimedia program unit of data that resides in system memory. Contrast with *static resource* .

E

earcon - An icon with an audio enhancement, such as a ringing telephone.

ECB - Event control block.

ECU - Script abbreviation for *extreme close-up* .

edit decision list (EDL) - Synonym for *edit list* .

edit list - A list of the specific video footage, with time-code numbers, that will be edited together to form the program. It is completed during the offline edit and used during the online edit. Synonym for *edit decision list (EDL).*

edit master - The final videotape from which all copies are made. See also *glass master* .

editing - Assembling various segments into the composite program.

EDL - Edit decision list.

EGA - Enhanced graphics adapter.

element - (1) A file or other stored data item. (2) An individual file that is part of a RIFF compound file. An element of a compound file also could be an entire RIFF file, a non-RIFF file, an arbitrary RIFF chunk, or arbitrary binary data. (3) The particular resource within a subarea that is identified by an element address.

emphasis - Highlighting, color change, or other visible indication of the condition of an object or choice and the effect of that condition on a user's ability to interact with that object or choice. Emphasis can also give a user additional information about the state of an object or choice.

encode - To convert data by the use of a code in such a manner that reconversion to the original form is possible. Contrast with *decode* .(T)

enhanced graphics adapter (EGA) - A graphics controller for color displays. The pel resolution of an enhanced graphics adapter is 3:4.

entry field - An area into which a user places text. Its boundaries are usually indicated.

erasable optical discs - Optical discs that can be erased and written to repeatedly.

establishing shot - In videotaping, a long shot used in the beginning of a program or segment to establish where the action is taking place and to give the sense of an environment.

EVCB - Event control block.

event - An occurrence of significance to a task; for example, the completion of an asynchronous operation, such as I/O.

event control block (ECB or EVCB) - A control block used to represent the status of an event.

event queue - In computer graphics, a queue that records changes in input devices such as buttons, valuators, and the keyboard. The event queue provides a time-ordered list of input events.

event semaphore - (1) Used when one or more threads must wait for a single event to occur. (2) A blocking flag used to signal when an event has occurred.

explicit event - An event supported by only some handlers, such as a custom event unique to a particular type of data.

EXT - Script abbreviation for *exterior* .

extended selection - A type of selection optimized for the selection of a single object. A user can extend selection to more than one object, if required. The two kinds of extended selection are contiguous extended selection and discontiguous extended selection.

extreme close-up - The shot obtained when the camera is positioned to show only the face or a single feature of the subject; in the case of an object, the camera is close enough to reveal an individual part of the object clearly.

F

facsimile machine - A functional unit that converts images to signals for transmission over a telephone system or that converts received signals back to images.

fade - To change the strength or loudness of a video or audio signal, as in "fade up" or "fade down."

fast threads - Threads created by an application that provide minimal process context, for example, just stack, register, and memory. With the reduced function, fast threads can be processed quickly.

FAX machine - Synonym for facsimile machine.

file cleanup - The removal of superfluous or obsolete data from a file.

file compaction - Any method of encoding data to reduce its required storage space.

file element - An individual file that is part of a RIFF compound file. An element of a compound file can also be an entire RIFF file, a non-RIFF file, an arbitrary RIFF chunk, or arbitrary binary data. See *media element* .

file format - A language construct that specifies the representation, in character form, of data objects in a file. For example, MIDI, M-Motion, or AVC.

file format handler - (1) I/O procedure. (2) Provides functions that operate on the media object of a particular data format. These functions include opening, reading, writing, seeking, and closing elements of a storage system.

file format IOProc - (1) An *installable I/O procedure* that is responsible for all technical knowledge of the format of a specific type of data, such as headers and data compression schemes. A file format IOProc manipulates multimedia data at the element level. A file format IOProc handles the element type it was written for and does not rely on any other file format IOProcs to do any processing. However, a file format IOProc might need to call a *storage system IOProc* to obtain data within a file containing multiple file elements. (2) See *IOProc* . (3) See also *storage system IOProc* .

filter - A certain type of node that modifies messages and forwards them. Filters are used to perform real-time processing of MIDI data. When a filter receives a message, it may perform some manipulation on it. It will then forward the message. (RTMIDI-specific term)

final script - The finished script that will be used as a basis for shooting the video. Synonym for *shooting script* .

first draft - A rough draft of the complete script.

first generation - In videotaping, the original or master tape; not a copy.

flashback - Interruption of chronological sequence by interjection of events occurring earlier.

flush stop - In data streaming, requests that the source stream handler be stopped but the target stream handler continue until the last buffer held at the time the stop was requested is consumed by the target stream handler.

flutter - A phenomenon that occurs in a videodisc freeze-frame when both video fields are not identically matched, thus creating two different pictures alternating every 1/60th of a second.

fly-by - Animation simulating a bird's-eye view of a three-dimensional environment.

fly-in - A DVE where one picture "flies" into another.

folder - A file used to store and organize documents or electronic mail.

footage - The total number of running feet of film used (as for a scene).

foreground image - The part of a display image that can be changed for every transaction. Contrast with *background image* .

form overlay - A pattern such as a report form, grid, or map used as background for a display image.

form type - A field in the first four bytes of the data field of a RIFF chunk. It is a four-character code identifying the format of the data stored in the file. A RIFF form is a chunk with a chunk ID of RIFF. For example, waveform audio files (WAVE files) have a form type of WAVE.

Format 0 MIDI file - All MIDI data is stored on a single track.

Format 1 MIDI file - All MIDI data is stored on multiple tracks.

forward - To re-transmit a message that was received. Each instance can have any number of links from it. When an instance receives a message, it may decide to send the same message along its links. This is known as forwarding. (RTMIDI-specific term)

four-character code (FOURCC) - A 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters (padded on the right with blank characters). Four-character codes are unique identifiers that represent the file format and I/O procedure.

FOURCC - Four-character code.

fps - Frames per second.

frame - In film, a complete television picture that is composed of two scanned fields, one of the even lines and one of the odd lines. In the NTSC system, a frame has 525 horizontal lines and is scanned in 1/30th of a second.

frame-step recording - Refers to the capturing of video and audio data frame by frame, from a computer-controlled, frame-steppable video source device, or a previously recorded AVI file.

frame-accurate searches - The ability of a videodisc player to play or search to specific frames on the videodisc via software or remote control. This capability is available on all CAV players, but currently only on a few CLV players. Most CLV players can only search or play to within one second (30 frames for NTSC or 25 frames for PAL).

frame animation - A process where still images are shown at a constant rate. See also *cast animation* .

frame grabber - A device that digitizes video images.

frame number - The number used to identify a frame. On videodisc, frames are numbered sequentially from 1 to 54,000 on each side and can be accessed individually; on videotape, the numbers are assigned by way of the SMPTE time code.

frame rate - The speed at which the frames are scanned-30 frames a second in NTSC video, 25 frames a second in PAL video, and 24 frames a second in *most* film.

A complete television picture frame is composed of two scanned fields, one of the even lines and one of the odd lines. In the NTSC system, a frame has 525 horizontal lines and is scanned in 1/30th of a second. In the PAL system, a frame has 625 horizontal lines and is scanned in 1/25th of a second.

freeze - Disables updates to all or part of the video buffer. The last video displayed remains visible. See also *unfreeze* .

freeze-frame - A frame of a motion-picture film that is repeated so as to give the illusion of a still picture.

full-frame time code - (1) A standardized method, set by the Society of Motion Picture and Television Engineers (SMPTE), of address coding a videotape. It gives an accurate frame count rather than an accurate clock time. (2) Synonym for *nondrop time code* .

full-motion video - Video playback at 30 frames per second for NTSC signals or 25 frames per second for PAL signals.

G

game port - On a personal computer, a port used to connect devices such as joysticks and paddles.

GDT - Global Descriptor Table.

general purpose interface bus - An adapter that controls the interface between the PC, Personal Computer XT*, or Personal Computer AT* and, for example, the InfoWindow display; also known as the IBM IEEE 488.

genlock - A device that comes on an adapter or plugs into a computer port and provides the technology to overlay computer-generated titles and graphics onto video images. It does this by phase-*lock* ing the sync *gen* eration of two video signals together so they can be merged. Genlock also converts a digital signal to NTSC or PAL format. (For example, flying logos and scrolling text on television shows are overlaid using a genlock.)

glass master - The final videodisc format from which copies are made.

Global Descriptor Table (GDT) - Defines code and data segments available to all tasks in an application.

GOCA - Graphic Object Content Architecture.

Graphic Object Content Architecture (GOCA) - (1) A data format for multimedia products. (2) A push button with graphic, two-state, and animation capabilities.

graphics overlay - The nature of dual plane video systems makes it possible to place graphics over video. Only the pels of the designated transparent color allows the video to show through. All other graphics pels appear on top of the video. Note that the video still exists in the video buffer under the non-transparent graphics pels.

graphics plane - In a dual plane video system, the graphics plane contains material drawn or generated by the graphics adapter. The graphics plane will be combined with the video plane to create an entire display image.

grayscale - See *greyscale.*

greyscale - When video is displayed in shades of black and white.

grouping - For Media Control Interface devices, refers to the ability to associate dissimilar devices for a common purpose. Grouping MCI

devices aids resource management by insuring that all devices in a group are kept together.

H

handshaking - The exchange of predetermined signals when a connection is established between two data set devices.

help view - A view of an object that provides information to assist users in working with that object.

Hi8 - High-band 8mm videotape format.

HID - Handler identification.

High Sierra Group (HSG) - (1) A group that set the standards for information exchange for a CD-ROM. (2) HSG also refers to those standards (HSG standards).

HMS - (1) Hours-minutes-seconds. (2) A time format for videodisc players.

HMSF - (1) Hours-minutes-seconds-frames. (2) A time format for videodisc players.

hot spot - The area of a display screen that is activated to accept user input. Synonym for *touch area* .

HSG - High Sierra Group.

HSI - Hue saturation intensity.

hue - Describes the position of a color in a range from blue to green. A hue level of 0 is maximum blue. A hue level of 100 is maximum green. Synonym for *tint* .

hue saturation intensity (HSI) - A method of describing color in three-dimensional color space.

HWID - Hardware identifier.

hypermedia - Navigation or data transfer between connected objects of different media types. For example, a user might navigate from an image object to an audio object that describes the image over a hypermedia link. Or, a representation of a graph object might be embedded in a document object with a hypermedia data transfer link, such that when the graph object is changed, the representation of that object in the document is also changed.

Hytime - An ANSI standard proposal that addresses the synchronization and linking of multimedia objects.

I

IDC - Inter-device communication mechanism provided by the OS/2 function ATTACHDD DevHelp.

identifier - (1) A sequence of bits or characters that identifies a program, device, or system to another program, device, or system. (2) In the C language, a sequence of letters, digits, and underscores used to identify a data object or function. (3) See *four-character code (FOURCC)* .

IDOCA - Integrated Data Object Content Architecture.

image - An electronic representation of a video still.

image bitsperpel - Pertaining to the number of colors supported by the current pel format. The currently accepted standard values are those supported by OS/2 bitmaps, for example, 1, 4, 8, or 24 bits per pel. In addition, 12 bits per pel formats are accepted for YUV images (including the 'yuvb' pel format for RDIB files).

image buffer - A location in memory where video images are stored for later use.

image buffer formats - The format or representation of data buffers containing video images.

image compression - The method of compressing video image data to conserve storage space.

image file format - The format or representation of data files containing video images.

Image Object Content Architecture (IOCA) - A data format for multimedia products.

image pelformat - Indicates the color representation that is to be used for images that are captured and saved. This normally includes palettized RGB, true-color RGB, or YUV color formats.

image quality - Represents the user's or application's subjective evaluation of complexity and quality of the image to be captured or saved. This setting is used to determine specific compression methods to use for saving the image.

implicit event - An event that all stream handlers always must support, such as *end of stream* or *preroll complete*).

in-betweening - Synonym for *tweening* .

in frame - Refers to a subject that is included within the frame. See also *frame* .

InfoWindow system - A display system that can combine text, graphics, and video images on a single display. The minimum system configuration is the IBM InfoWindow Color Display, a system unit, a keyboard, and one or two videodisc players.

input locking mask - A filter, or mask, that controls which areas of the display can display or freeze video.

input/output control (IOCtl) - A system function that provides a method for an application to send device-specific control commands to a device driver.

installable I/O procedure - A file format handler that provides functions that operate on the media object of a particular data format. These functions include opening, reading, writing, seeking, and closing elements.

instance - See *node instance* . (RTMIDI-specific term)

INT - Script abbreviation for *interior* .

Integrated Data Object Control Architecture (IDOCA) - A data format for multimedia products.

interactive multimedia - The delivery of information content through combinations of video, computer graphics, sound, and text in a manner that allows the user to interact.

interactive program - A running program that can receive input from the keyboard or another input device. Contrast with *noninteractive program* .

interactive videodisc system (IVS) - A system in which a user can interact with a videodisc display image by entering commands to the computer through a device such as a keyboard or keypad or by touching a touch-sensitive screen at specific points on the display surface.

interlace flicker - The apparent flicker when one field of an interlaced image contains more light than the other field due to the placement of image details with respect to the separate fields. Two methods used to avoid interlace flicker: limit the vertical resolution on natural images (as opposed to text or graphics); design characters so that each character has an equal number of pels in each field.

interlacing - In multimedia applications, a characteristic of video image display that results in greater image clarity. In effect, the video image is traced across the screen twice. (The time delay between the two tracings makes this effect undesirable for normal computer-generated graphics.) Synonymous with *interleaving* .

interleaving - (1) The simultaneous accessing of two or more bytes or streams of data from distinct storage units. (2) The alternating of two or more operations or functions through the overlapped use of a computer facility. (3) In a duplicator, the process of inserting absorbent sheets between successive sheets of the copy paper to prevent set-off. (T) (4) Synonym for interlacing.

internal I/O procedure - An I/O procedure that is built in to the MMPM/2 system, including DOS, MEM, BND, and CF IOProcs.

International Organization for Standardization (ISO) - An organization of national standards bodies from various countries established to promote development of standards to facilitate international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

IOCtl - Input/output control.

IOProc - A file format handler that provides functions that operate on the media object of a particular data format. These processes include opening, reading, writing, seeking, and closing elements of a storage system. There are two classes of I/O procedures: *file format* and *storage system* .

iris - To fade a picture by operating the iris (aperture) on the camera in a certain way; the type of computer image dissolve accomplished by operating the aperture in a certain way.

ISO - International Organization for Standardization.

ISP - IBM Signal Processor. An IBM proprietary digital signal processor.

ISPOS - IBM Signal Processor Operating System.

ISV - Independent software vendor.

items - Options, choices or keywords such as one or more of the following options, choices or keywords can or should be specified. Sometimes use of these items can also be exclusive or some items may not be compatible with other items.

IVS - Interactive videodisc system.

J

JIT - Just in time. (Often used with "learning" as "JIT learning".) Multimedia help functions, closely integrated with applications, that employ voice output to guide the user.

Joint Photographic Experts Group (JPEG) - A group that is working to establish a standard for compressing and storing still images in digital form. JPEG also refers to the standard under development by this group (JPEG standard).

joy stick - In computer graphics, a lever that can pivot in all directions and that is used as a locater device. (Resembles an airplane's joy stick).

JPEG - Joint Photographic Experts Group.

K

keeper - Synonym for *buy*.

kernel - (1) The part of an operating system that performs basic functions such as allocating hardware resources. (2) A program that can run under different operating system environments. (3) A part of a program that must be in main storage in order to load other parts of the program.

key frames - The start and end frames of a single movement in an animation sequence; also can refer to the periodic full-frame image interspersed in the stream to allow random starts from these full-frame images (key frames).

keypad - A small, often hand-held, keyboard.

L

laser - Light amplification by stimulated emission of radiation; the device that produces this light.

latency - In video, the time it takes for the light from the phosphor screen to decay after the excitation is removed. Long-persistence phosphor has less flicker of still images, but more blurring of moving images.

level one videodisc applications - Interactive applications based on manual keypad functions, picture stops, and chapter stops.

level three videodisc applications - Interactive applications controlled by an external computer that uses the videodisc player as a peripheral device.

level two videodisc applications - Interactive applications controlled by the keypad and the videodisc player's internal computer. The control program is recorded on the videodisc itself.

LIB - Dynamic-link definition library. A file containing the data needed to build a program .EXE file but which does not contain the dynamic-link programs themselves. Contrast with *dynamic-link library* .

light pen - A light-sensitive pick device that is used by pointing it at the display surface.

line graphics - Synonym for *coordinate graphics* .

line pairing - A faulty interlace pattern in which the lines of the second field begin to pair with the lines of the first field rather than fit exactly within them.

linear audio - The analog audio on the linear track of videotape that can be recorded without erasing existing video; used for audio dubbing after video is edited.

linear video - A sequence of motion footage played from start to finish without stops or branching, like a movie.

link - A one-way link (directed edge) from one instance to another. If an instance wishes to send a message, it is sent along all the links from it. An instance can have any number of links coming from it, and any number of other instances can have links to it. An instance cannot select along which links the message should be sent. See also *slot* . (RTMIDI-specific term)

LIST chunk - A chunk that contains a list or an ordered sequence of subchunks.

list type - (1) A field in the first four bytes of the data field of a LIST chunk. (2) A four-character code identifying the contents of the list.

locked memory - An area of memory that is not available for use because it is being held by another process.

long shot - (1) A camera angle that reveals the subject and the surroundings. Often used as an establishing shot. (2) Synonym for *wide shot* .

LS - Script abbreviation for *long shot* .

luminance signal - The portion of image information that provides brightness. Alone, luminance provides a monochrome image.

M

M-ACPA - M-Audio Capture and Playback Adapter.

M-Audio Capture and Playback Adapter (M-ACPA) - An adapter card (for use with the IBM PS/2 product line) that provides the ability to record and play back high quality sound. The adapter converts the audio input (analog) signals to a digital format that is compressed and stored for later use.

master stream handler - Controls the behavior of one or more subordinate objects (the *slave streams*).

matte - In film, an opaque piece of art or a model that leaves a selected area unexposed to be filled on a subsequent pass or in composite.

MCD - Media control driver.

MCI - Media Control Interface.

M-Control Program/2 - The software interface required for the M-Motion Video Adapter/A. It consists of APIs or toolkits for DOS, Windows, Windows MCI, OS/2, and OS/2 MMPM/2. It also includes Pioneer and Sony videodisc player drivers for these environments. See also *M-Motion Video Adapter/A* .

MDM - Media device manager.

media - More than one hardware medium.

media component - A processor of audiovisual information or media. Media components can be either internal or external physical devices or defined mechanisms for effecting higher-level function from internal hardware and software subsystems. (An example is a waveform player component that utilizes the DSP subsystem and data streaming services to effect audio playback functions.)

media component capabilities - The functionality of a media component, including component functions that are supported.

media component type - A class of media components that exhibit similar behavior and capabilities. Examples of media component types are analog video display hardware and MIDI synthesizers.

media control driver (MCD) - A software implementation or method that effects the function of a media component. For OS/2, a media control driver or *media driver* is a dynamic-link library (or set of libraries) that utilizes physical device drivers, Media Device Manager services, and OS/2 to implement the function of the media component.

Media Control Interface (MCI) - A generalized interface to control multimedia devices. Each device has its own MCI driver that implements a standard set of MCI functions. In addition, each media driver can implement functions that are specific to the particular device.

media device - A processor of audiovisual information or media. Media components can be either internal or external physical devices or defined mechanisms for effecting higher-level function from internal hardware and software subsystems. (An example is a waveform player component that utilizes the DSP (Digital Signal Processor) subsystem and data-streaming services to effect audio-playback functions.)

media device capabilities - The functionality of a media component, including supported component functions.

media device connection - A physical or logical link between media component connectors for a particular set of media component instances.

media device connector - A physical or logical input or output on a media component.

media device connector index - An identifier for a media component connector.

media device ID - Media component identification. A unique identifier for a component.

media device instance - A case of an application's use of a media component.

media device manager (MDM) - A system service that, when two or more applications attempt to control a media device, determines which process gains access.

media driver - A device driver for a multimedia device. See also *device driver* .

media driver - A software implementation or method that effects the function of a media device.

media element manager (MEM) - A system service that manipulates multimedia data.

media programming interface (MPI) - A subsystem that provides a comprehensive system programming API layer for multimedia applications.

media segment - An audiovisual object of some type, such as a waveform, song, video clip, and so on.

media unit - A medium on which files are stored; for example, a diskette.

media volume - A (possibly heterogeneous) physical or logical collection of media segments. (Examples are a videodisc, video tape, compact audio disc, OFF file, and RIFF file.)

media volume file - A media volume that is embodied as a conventional binary computer file within a computer file system on storage devices such as disks, diskettes, or CD-ROMs. Such storage devices can be either local or remote. Media volume files can be of various formats, such as OFF or RIFF, and contain segments of various types.

medium shot - A camera angle that reveals more of the subject than a close-up but less than a wide shot, usually from face to waistline; sometimes called a *mid-shot* .

MEM - Media element manager.

MEM IOProc - An internal I/O procedure provided by the MMPM/2 system that supports memory files.

memory file - A block of memory that is perceived as a file by an application.

memory playlist - A data structure in the application used to specify the memory addresses to play from or record to. The application can modify the playlist to achieve various effects in controlling the memory stream.

message interface - See *command message interface* .

MIDI Mapper - Provides the ability to translate and redirect MIDI messages to achieve device-independent playback of MIDI sequences.

MIDI message - A sequence of bytes that conform to the MIDI standard. There are two categories: System Exclusive (SysEx) and non-SysEx messages. SysEx messages can be of any length, whereas non-SysEx messages are between one and three bytes.

mid-shot - See *medium shot* .

millisecond - One thousandth of a second.

minutes-seconds-frames (MSF) - A time format based on the 75-frames-per-second CD digital audio standard.

MIPS - Millions of instructions per second. A unit of measure of processing performance equal to one million instructions per second.

mix - The combination of audio or video sources during postproduction.

mixed-media system - Synonym for *multimedia system* .

Mixed Object : Document Content Architecture (MO:DCA) - A data format for multimedia products.

mixer - A device used to simultaneously combine and blend several inputs into one or two outputs.

mixing - (1) In computer graphics, the result of the intersection of two or more colors. (2) In filming, the combining of audio and video sources that is accomplished during postproduction at the *mix* . (3) In recording, the combining of audio sources.

MMIO - Multimedia input/output.

MMIO file services - System services that enable an application to access and manipulate multimedia data files.

MMIO manager - Multimedia input/output manager. The MMIO manager provides services to find, query, and access multimedia data objects. It also supports the functions of memory allocation and file compaction. The MMIO manager uses IOProcs to direct the input and output associated with reading from and writing to different types of storage systems or file formats.

M-Motion - A multimedia platform that offers analog video in addition to quality sound and images. The M-Motion environment consists of the M-Motion Video Adapter/A and the M-Control Program/2 master stream handler.

M-Motion Video Adapter/A - (1) A Micro Channel adapter that receives and processes signals from multiple video and audio sources, and then sends these signals to a monitor and speakers. (2) Dual plane video hardware that offers analog video in addition to quality sound and images. (3) The M-Motion Video Adapter/A requires the M-Control Program/2.

MMPM/2 - Multimedia Presentation Manager/2. See *OS/2 multimedia* .

MMTIME - Standard time and media position format supported by the media control interface. This time unit is 1/3000 second, or 333 microseconds.

MO:DCA - Mixed Object : Document Content Architecture.

mode - A method of operation in which the actions that are available to a user are determined by the state of the system.

model - The conceptual and operational understanding that a person has about something.

module - A language construct that consists of procedures or data declarations and that can interact with other constructs.

moire - An independent, usually shimmering pattern seen when two geometrically regular patterns (as a sampling frequency and a correct frequency) are superimposed. The moire pattern is an alias frequency. See also *aliasing* .

monitor - See *video monitor* .

monitor window - A graphical window, available from a digital video device, which displays the source rectangle, and any subset of this video capture region. See *destination rectangle* for related information.

motion-control photography - A system for using computers to precisely control camera movements so that the different elements of a shot-models and backgrounds, for example-can later be composited with a natural and believable unity.

motion video capture adapter - An adapter that, when attached to a computer, allows an ordinary television picture to be displayed on all or part of the screen, mixing high-resolution computer graphics with video; also enables a video camera to become an input device.

Motion Video Object Content Architecture (MVOCA) - A data format for multimedia products.

Moving Pictures Experts Group (MPEG) - A group that is working to establish a standard for compressing and storing motion video and animation in digital form.

MPEG - Moving Pictures Experts Group.

MPI - Media Programming Interface.

MPI application services - Media Programming Interface application services. Functional services provided by MPI to application programs and higher-level programming constructs, such as multimedia controls.

MS - Script abbreviation for *medium shot* .

MSF - Minutes-seconds-frames.

multimedia - Material presented in a combination of text, graphics, video, image, animation, and sound.

multimedia data object - In an application, an element of a data structure (such as a file, an array, or an operand) that is needed for program execution and that is named or otherwise specified by the allowable character set of the language in which the program is coded.

Multimedia File I/O Services - System services that provide a generalized interface to manipulate multimedia data. The services support buffered and unbuffered file I/O, standard RIFF files, and installable I/O procedures.

multimedia input/output (MMIO) - (1) System services that provide a variety of functions for media file access and manipulation. (2) A consistent programming interface where an application, media driver, or stream handler can refer to multimedia files, read and write data to the files, and query the contents of the files, while remaining independent of the underlying file formats or the storage systems that contain the files.

multimedia navigation system - A tool that gives the information product designer the freedom to link various kinds and pieces of data in a variety of ways so that users can move through it nonsequentially.

multimedia system - (1) A system capable of presenting multimedia material in its entirety. (2) Synonym for *mixed-media system* .

multiple selection - A selection technique in which a user can select any number of objects, or not select any.

Musical Instrument Digital Interface (MIDI) - A protocol that allows a synthesizer to send signals to another synthesizer or to a computer, or a computer to a musical instrument, or a computer to another computer.

mute - To temporarily turn off the audio for the associated medium.

mux - An abbreviation for multiplexer. See also *mixer* .

MVOCA - Motion Video Object Content Architecture.

N

NAPLPS - North American Presentation Level Protocol Syntax.

National Television Standard Committee (NTSC) - A committee that set the standard for color television broadcasting and video in the United States (currently in use also in Japan); also refers to the standard set by this committee (NTSC standard).

node - An abstract term indicating either a node class or a node instance. When used with a class qualifier (for example, application node) it implies an instance (for example, instance of an application class). (RTMIDI-specific term)

node class - A definition of the behavior of a node instance. All instances of the same class are expected to have the same behavior and purpose, although this restriction is not enforced by the driver. (RTMIDI-specific term)

node instance - A vertex in the node network that can receive and transmit MIDI messages. (RTMIDI-specific term)

node network - The collection (graph) of node instances and links. (RTMIDI-specific term)

nondrop time code - Synonym for *full-frame time code* .

noninteractive program - A running program that cannot receive input from the keyboard or other input device.

non-streaming device - (1) A device that contains both source and destination information for multimedia. (2) A device that transmits data (usually analog) directly, without streaming to system memory.

North American Presentation Level Protocol Syntax (NAPLPS) - A protocol used for display and communication of text and graphics in a videotex system; a form of vector graphics.

notebook - A graphical representation that resembles a perfect-bound or spiral-bound notebook that contains pages separated into sections by tabbed divider pages. A user can turn the pages of a notebook to move from one section to another.

NTSC - National Television Standard Committee.

null streaming - (1) The behavior of a stream that can be created and started but which has no associated data flow. (2) The behavior of a stream that can be created and started but which has no associated data flow. For example, a CD-DA is a non-streaming device. (3) A

device that does not stream its data through the MMPM/2 streaming system For example, a CDDA device is a non-streaming device.

O

object - (1) Anything that exists in and occupies space in storage and on which operations can be performed; for example, programs, files, libraries, and folders. (2) Anything to which access is controlled; for example, a file, a program, an area of main storage. (3) See also *data object* and *media object* .

object-action paradigm - A method where users select the object that they want to work with, then choose the action they wish to perform on that object. See *object orientation* .

object class - A categorization or grouping of objects that share similar behaviors and characteristics.

object connection - A link between two objects. Connections can be used for navigation, as with *hypermedia,* or for data transfer between objects.

Object Content Architecture (OCA) - A data format for multimedia products.

object decomposition - The process of breaking an object into its component parts.

object orientation - An orientation in a user interface in which a user's attention is directed toward the objects the user works with, rather than applications, to perform a task.

object-oriented user interface - A type of user interface that implements object orientation and the object-action paradigm.

object template - An object that can be used to create another object of the same object class. The template is a basic framework of the object class, and the newly created object is an instance of the object class.

OCA - Object Content Architecture.

OEM - Original equipment manufacturer.

OFF - Operational file format.

offline edit - A preliminary or test edit usually done on a low-cost editing system using videocassette work tapes. (An offline edit is done so that decisions can be made and approvals given prior to the final edit.)

online edit - The final edit, using the master tapes to produce a finished program.

operational file format (OFF) - A file format standard.

optical disc - A disc with a plastic coating on which information (as sound or visual images) is recorded digitally as tiny pits and read using a laser. The three categories of optical discs are CD-ROM, WORM, and erasable.

optical drive - Drives that run optical discs.

optical reflective disc - A designation of the means by which the laser beam reads data on an optical videodisc. In the case of a reflective disc, the laser beam is reflected off a shiny surface on the disc.

opticals - Visual effects produced optically by means of a device (an optical printer) that contains one camera head and several projectors. The projectors are precisely aligned so as to produce multiple exposures in exact registration on the film as in the camera head.

original footage - The footage from which the program is constructed.

OS/2 multimedia - A subsystem service of OS/2 that provides a software platform for multimedia applications. It defines standard interfaces between multimedia devices and OS/2 multimedia applications.

overlay - The ability to superimpose text and graphics over video.

overlay device - Provides support for video overlaying along with video attribute elements. The video overlaying handles tasks such as displaying, and sizing video. Synonym for *video overlay device* .

P

PAL - Phase Alternation Line.

palette - See *color palette* , *standard palette* , and *custom palette* .

pan - A camera movement where the camera moves sideways on its stationary tripod; left-to-right balance in an audio system.

panel - A particular arrangement of information grouped together for presentation to users in a window.

panning - Progressively translating an entire display image to give the visual impression of lateral movement of the image.

In computer graphics, the viewing of an image that is too large to fit on a single screen by moving from one part of the image to another.

paradigm - An example, pattern, or model.

patch mapping - The reassignment of an instrument patch number associated with a specific synthesizer to the corresponding standard patch number in the General MIDI specification.

pause - To temporarily halt the medium. The halted visual should remain displayed but no audio should be played.

pause stop - In data streaming, a stop that pauses the data stream but does not disturb any data.

PDC - Physical device component.

PDD - Physical device driver.

pedestal up/down - A camera movement where the camera glides up or down on a boom.

pel - The dimensions of a toned area at a picture element. See also *picture element* .

Phase Alternation Line (PAL) - Television broadcast standard for European video outside of France and the Soviet Union.

physical device driver (PDD) - A program that handles hardware interrupts and supports a set of input and output functions.

picon - A graphic or natural image reduced to icon size. Similar to *thumbnail.*

picture element - In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity. See also *pel* .

picture-in-picture - A video window within a larger video window.

pixel - See *picture element* .

plaintext - Nonencrypted data.

platform - In computer technology, the principles on which an operating system is based.

play backward - To play the medium in the backward direction.

playback window - The graphic window in which software motion video is displayed. This window can be supplied by an application, or a default window can be created by a digital video device.

play forward - To play the medium in the forward direction.

pointer - A symbol, usually in the shape of an arrow, that a user can move with a pointing device. Users place the pointer over objects they want to work with.

pointing device - A device, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

polish - The version of the script submitted for final approval.

polyphony - A synthesizer mode where more than 1 note can be played at a time. Most synthesizers are 16-note to 32-note polyphonic.

postproduction - The online and offline editing process.

PPQN - (1) Parts-per-quarter-note. (2) A time format used in musical instrument digital interface (MIDI).

preproduction - The preparation stage for video production, when all logistics are planned and prepared.

preroll - The process of preparing a device to begin a playback or recording function with minimal latency. During a multimedia sequence, it might require that two devices be cued (prerolled) to start playing and recording at the same time.

primary window - A window in which the main interaction between a user and an object takes place.

Proc - A custom procedure, called by the particular utility manager, to handle input or output to files of a format different from DOS, MEM, or BND; for example, AVC or TIFF. By installing custom procedures, existing applications no longer need to store multiple copies of the same media file for running on various platforms using different file formats. See also *static resource* and *dynamic resource* .

production - In videotaping, the actual shooting.

production control room - The room or location where the monitoring and switching equipment is placed for the direction and control of a television production.

progress indicator - A control, usually a read-only slider, that informs the user about the status of a user request.

props - In videotaping, support material for the shoot, for example, equipment being promoted, auxiliary equipment, software, or supplies; anything provided to make the set look realistic and attractive.

protection master - A copy of the edit master that is stored as a backup.

PS/2 CD-ROM-II Drive - An IBM CD-ROM drive that can play compact disc digital audio (CD-DA) and CD-ROM/XA interleaved audio, video, and text, and adheres to the Small Computer System Interface (SCSI). The drive can be installed on Micro Channel and non-Micro Channel IBM PS/2 systems.

pulse code modulation (PCM) - In data communication, variation of a digital signal to represent information.

push button - A graphical control, labeled with text, graphics, or both, that represents an action that will be initiated when a user selects it. For example, when a user clicks on a *Play* button, a media object begins playing.

R

raster graphics - Computer graphics in which a display image is composed of an array of pels arranged in rows and columns.

raw footage - Synonym for *original footage* .

ray-tracing - A technique used by 3-D rendering software programs that automatically figures an object's position in three dimensions and calculates shadows, reflections, and hidden surfaces based on user-entered light locations and material characteristics. (In other words, if the user orders an object to be a mirror, the computer produces the mirror with all its correct reflective properties.)

real time - (1) Pertaining to the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process. This term is also used to describe systems operating in conversational mode and processes that can be influenced by human intervention while they are in progress. (2) A process control system or a computer-assisted instruction program, in which response to input is fast enough to affect subsequent output.

real-time recording - Refers to the capturing of video and audio data in real time, as the analog signals are generated from the video source device. The video source device can be a camcorder, or a videotape or videodisc player.

record - To transfer data from one source (for example, microphone, CD, videodisc) or set of sources to another medium.

Redbook audio - The storage format of standard audio CDs. See also *compact disc, digital audio (CD-DA)* .

reference frame - (1) Refers to the complete frame that is created at periodic intervals in the output stream. An editing operation always begins at a reference frame. (2) Synonymous with *key frame* and *I-frame* . (3) See *delta frame* .

reflective disc - See *optical reflective disc* .

render - In videotaping, to create a realistic image from objects and light data in a scene.

repeat - A mode which causes the medium to go to the beginning and start replaying when it reaches the medium's end.

resolution - (1) In computer graphics, a measure of the sharpness of an image, expressed as the number of lines and columns on the display screen or the number of pels per unit of area. (2) The number of lines in an image that an imaging system (for example, a

telescope, the human eye, a camera, and so on) can resolve. A higher resolution makes text and graphics appear clearer.

resource - As used in the multimedia operating system, any specific unit of data created or used by a multimedia program. See also *static resource* and *dynamic resource* .

resource handler - A system service that loads, saves, and manipulates multimedia program units of data.

resource interchange file format (RIFF) - A tagged file format framework intended to be the basis for defining new file formats.

resync - Recovery processing performed by sync-point services when the failure of a session, transaction program, or LU occurs during sync-point processing. The purpose of resync is to return protected resources to consistent states.

resync tolerance value - A minimum time difference expressed in MMTIME format.

Revisable Form Text : Document Content Architecture (RFT:DCA) - A data format for multimedia products.

rewind - To advance the medium in the backward direction quickly, and optionally allow the user to scan the medium.

RFT:DCA - Revisable Form Text : Document Content Architecture.

RGB - Color coding where the brightness of the additive primary colors of light, red, green, and blue, are specified as three distinct values of white light.

RIFF - Resource interchange file format.

RIFF chunk - A chunk with a chunk ID of *RIFF* . In a RIFF file, this must be the first chunk.

RIFF compound file - A file containing multiple file elements or one file element that makes up the entire RIFF file. The MMIO manager provides services to find, query, and access any file elements in a RIFF compound file. Synonym for *bundle file* .

rotoscope - A camera setup that projects live-action film, one frame at a time, onto a surface so that an animator can trace complicated movements. When filmed, the completed animation matches the motion of the original action.

rough cut - (1) The result of the offline edit. (2) A video program that includes the appropriate footage in the correct order but does not include special effects.

RTMIDI - Real-time MIDI subsystem.

S

SAA - Systems Application Architecture.

safety - An extra shot of a scene that is taken as a backup after an acceptable shot (the *buy*) has been acquired.

saturation - The amounts of color and grayness in a hue that affect its vividness; that is, a hue with high saturation contains more color and less gray than a hue with low saturation. See also *hue* .

sampler - A device that converts real sound into digital information for storage on a computer.

scan backward - To display the video and optionally play the audio while the medium is advancing in the backward direction rapidly.

scan converter - A device that converts digital signal to NTSC or PAL format.

scan forward - To display the video and optionally play the audio while the medium is advancing in the forward direction rapidly.

scan line - (1) In a laser printer, one horizontal sweep of the laser beam across the photoconductor. (2) A single row of picture elements.

scanner - A device that examines a spatial pattern, one part after another, and generates analog or digital signals corresponding to the pattern.

SCB - Subsystem control block.

scene - A portion of video captured by the camera in one continuous shot. The scene is shot repeatedly (each attempt is called a *take*) until an acceptable version, called the *buy*, is taken.

scheduler - The code responsible for passing messages to instances. The scheduler has two queues, one for normal real-time MIDI messages (Q1), and the other for low-priority SysEx messages (Q2). (RTMIDI-specific term)

scheduler daemon - A small executable program, MIDIDMON.EXE, which is used to provide deferred-interrupt processing for the scheduler. The daemon is a super high-priority thread which gets blocked in ring 0. When the scheduler needs to run, this thread is unblocked. When the scheduler is finished, it re-blocks itself. When the unblocking occurs during an interrupt, the OS/2 kernel runs the daemon thread immediately after the interrupt handler has exited. This approach guarantees that the scheduler runs at task time.

scripting - Writing needed dialog.

scroll bar - A window component that shows a user that more information is available in a particular direction and can be scrolled into view. Scroll bars should not be used to represent an analog setting, like volume. Sliders should be used.

SCSI - Small computer system interface.

SECAM - Sequential Couleurs a Memoire. The French standard for color television.

secondary window - A window that contains information that is dependent on information in a primary window and is used to supplement the interaction in the primary window.

secondary window manager - A sizable dialog manager that enables application writers to use CUA-defined secondary windows instead of dialog boxes.

second generation - A direct copy from the master or original tape.

selection - The act of explicitly identifying one or more objects to which a subsequent choice will apply.

selection technique - The method by which users indicate objects on the interface that they want to work with.

semaphore - (1) A variable that is used to enforce mutual exclusion. (T) (2) An indicator used to control access to a file; for example, in a multiuser application, a flag that prevents simultaneous access to a file.

sequencer - A digital tape recorder.

set - In videotaping, the basic background or area for production.

settings - Characteristics of objects that can be viewed and sometimes altered by the user. Examples of a file's settings include name, size, and creation date. Examples of video clip's settings include brightness, contrast, color, and tint.

settings view - A view of an object that provides a way to change characteristics and options associated with the object.

SFX - Script abbreviation for *special effects* .

shade - To darken with, or as if with, a shadow; to add shading to.

sharpness - Refers to the clarity and detail of a video image. A sharpness value of 0 causes the video to be generally fuzzy with little detail. A sharpness value of 100 causes the video to be generally very detailed, and may appear grainy.

SHC - Stream handler command.

shoot - To videotape the needed pictures for the production.

shooting script - Synonym for *final script* .

shot list - A list containing each shot needed to complete a production, usually broken down into a schedule.

simple device - A multimedia device model for hardware which does not require any additional objects, known as device elements, to perform multimedia functions.

sine wave - A waveform that represents periodic oscillations of a pure frequency.

single plane video system - Refers to when video and graphics are combined into one buffer. This may appear the same as a dual plane video system, but since all the data is in one buffer, capture and restore operations will obtain both graphics and video components in one operation. See also *dual plane video system* .

single selection - A selection technique in which a user selects one, and only one, item at a time.

slave stream - A stream that is dependent on the *master stream* to maintain synchronization.

slave stream handler - In SPI, regularly updates the sync pulse EVCB with the stream time. The Sync/Stream Manager checks the slave stream handler time against the master stream time to determine whether to send a sync pulse to the slave stream handler.

slider - A visual component of a user interface that represents a quantity and its relationship to the range of possible values for that quantity.

A user can also change the value of the quantity. Sliders are used for volume and time control.

slider arm - The visual indicator in the slider that a user can move to change the numerical value.

slider button - A button on a slider that a user clicks on to move the slider arm one increment in a particular direction, as indicated by the directional arrow on the button.

slide-show presentation - Synonym for *storyboard* .

slot - A distinct position in an instance from which links can be attached. The same message is sent along all links on a slot, but an instance can determine at run-time on which slots the message should be sent. An instance can support multiple slots if it wants to be able to send different messages to different targets. (RTMIDI-specific term)

small computer system interface (SCSI) - An input and output bus that provides a standard interface between the OS/2* multimedia system and peripheral devices.

SMH - Stream manager helper.

SMPTE - Society of Motion Picture and Television Engineers.

SMPTE time code - A frame-numbering system developed by SMPTE that assigns a number to each frame of video. The 8-digit code is in the form HH:MM:SS:FF (hours, minutes, seconds, frame number). The numbers track elapsed hours, minutes, seconds, and frames from any chosen point.

SMV - Software motion video.

socketable user interface - An interface defined by multimedia controls that enable the interface to be plugged into and unplugged from applications without affecting the underlying object control subsystem.

sound track - Synonym for *audio track* .

source node - An instance which can generate a compound message. Hardware nodes generate messages from data received from Type A drivers. Application nodes generate them from data sent from an application. (RTMIDI-specific term)

source rectangle - An abstract region representing the area available for use by a video capture adapter. This window is displayed in the monitor window of the digital video device. A subset of the maximum possible region to be captured can be defined; such a subset is shown by an animated dashed rectangle in the monitor window.

source window - See *source rectangle* .

SPCB - Stream protocol control block.

special effects - In videotaping, any activity that is not live footage, such as digital effects, computer manipulation of the picture, and nonbackground music.

SPI - Stream programming interface.

split streaming - A mechanism provided by the Sync/Stream Manager to create one data stream source with multiple targets.

SPP - A time format based on the number of beats-per-minute in the MIDI file.

sprite - An animated object that moves around the screen without affecting the background.

sprite graphics - A small graphics picture, or series of pictures, that can be moved independently around the screen, producing animated effects.

squeeze-zoom - A DVE where one picture is reduced in size and displayed with a full-screen picture.

SSM - Sync/Stream Manager.

standard multimedia device controls - These controls provide the application developer with a CUA compliant interface for controlling audio attributes, video attributes, and videodisc players. These controls simplify the programming task required to create the interface and handle the presentation of the interface and all interaction with the user. They also send the Media Control Interface (MCI) commands to the Media Device Manager (MDM) for processing.

standard objects - A set of common, cross-product objects provided and supported by the system. Examples include folders, printers, shredders, and media players.

standard palette - A set of colors that is common between applications or images. See also *custom palette* and *color palette* .

static resource - A *resource* that resides on any read-and-write or read-only medium. Contrast with *dynamic resource* .

status area - Provides information as to the state of the medium and device, or both. It should indicate what button is currently pressed and

what modes (for example, mute) are active.

step backward - To move the medium backward one frame or segment at a time.

step forward - To move the medium forward one frame or segment at a time.

still - A static photograph.

still image - See *video image* .

still video capture adapter - An adapter that, when attached to a computer, enables a video camera to become an input device. See also *motion video capture adapter* .

stop - Halt (stops) the medium.

storage system - The method or format a functional unit uses to retain or retrieve data placed within the unit.

storage system IOProc - A procedure that unwraps data objects such as RIFF files, RIFF compound files, and AVC files. IOProcs are ignorant of the content of the data they contain. A storage system IOProc goes directly to the OS/2 file system (or to memory in the case of a MEM file) and does not pass information to any other file format or storage system IOProc. The internal I/O procedures provided for DOS files, memory files, and RIFF compound files are examples of storage system IOProcs, because they operate on the storage mechanism rather than on the data itself. See also *file format IOProc* .

storyboard - (1) A visual representation of the script, showing a picture of each scene and describing its corresponding audio. (2) Synonym for *slide-show presentation* .

storyboarding - Producing a sequence of still images, such as titles, graphics, and images, to work out the visual details of a script.

stream - To send data from source to destination via buffered system memory.

stream connector - A port or connector that a device uses to send or receive. See also *connector* .

stream handler - A routine that controls a program's reaction to a specific external event through a continuous string of individual data values.

stream handler command (SHC) - Synchronous calls provided by both ring 3 DLL stream handlers as a DLL call and by ring 0 PDD stream handlers as a IDC call. The stream handler commands are provided through a single entry point, SHCEntryPoint, which accepts a parameter structure on input. This enables the DLL and PDD interfaces to the stream manager to be the same.

stream manager - A system service that controls the registration and activities of all stream handlers.

stream manager helper (SMH) - Routines provided by the stream manager for use by all stream handlers. The stream handlers use these helper routines to register with the manager, report events, and synchronize cues to the manager to request or return buffers to the manager. They are synchronous functions and are available to both ring 3 DLL stream handlers as a DLL call and to ring 0 PDD stream handlers.

stream programming interface - A system service that supports continual flow of data between physical devices.

stream programming interface (SPI) - A system service that supports continual flow of data between physical devices.

stream protocol control block (SPCB) - The system service that controls the behavior of a specified stream type. This enables you to subclass a stream's data type, change data buffering characteristics, and alter synchronization behavior and other stream events.

strike - In videotaping, to clear away, remove, or dismantle anything on the set.

subchunk - The first *chunk* in a RIFF file is a *RIFF chunk* ; all other chunks in the file are subchunks of the RIFF chunk.

subclassing - The act of intercepting messages and passing them on to their original intended recipient.

super - Titles or graphics overlaid on the picture electronically. See also *superimpose* .

superimpose - To overlay titles or graphics on the picture electronically.

S-video - (1) Separated video or super video. (2) A signal system using a Y/C format. (3) See also *Y/C* , *composite video* , and *component video* .

S-Video input connector - A special connector that separates the chrominance from the luminance signal.

sweetening - (1) The equalization of audio to eliminate noise and obtain the cleanest and most level sound possible. (2) The addition of laughter to an audio track.

switching - Electronically designating, from between two or more video sources, which source's pictures are recorded on tape. Switching can occur during a shoot or during an edit.

symmetric video compression - A technology in which the computer can be used to create, as well as play back, full-motion, full-color video.

sync - Synchronization or synchronized.

synchronization - The action of forcing certain points in the execution sequences of two or more asynchronous procedures to coincide in time.

synchronous - Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals.

sync group - A *master stream* and all its *slaves* that can be started, stopped, and searched as a group by using the slaves flag on each of the following SPI functions:

- SpiStartStream
- SpiStopStream
- SpiSeekStream

sync pulse - A system service that enables each slave stream handler to adjust the activity of that stream so that synchronization can be maintained. Sync pulses are introduced by transmission equipment into the receiving equipment to keep the two equipments operating in step.

sync signal - Video signal used to synchronize video equipment.

synthesizer - A musical instrument that allows its user to produce and control electronically generated sounds.

system message - A predefined message sent by the MMIO manager for the message's associated function. For example, when an application calls mmioOpen, the MMIO manager sends an MMIOM_OPEN message to an I/O procedure to open the specified file.

Systems Application Architecture (SAA) - A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

T

tagged image file format (TIFF) - An easily transportable image file type used by a wide range of multimedia software.

take - During the shoot in videotaping, each separate attempt at shooting a scene. This is expressed as: Scene 1, Take 1; Scene 1, Take 2, and so on.

talent - On-screen person (professional or amateur) who appears before the camera or does voice-over narration.

TAM - Telephone answering machine.

target node - An instance which receives a message but does not forward it because it is the final instance in a chain of processing. For example, a hardware node is a target node because when it receives a message, it sends it to another device driver, and not to another instance. (RTMIDI-specific term)

tearing - Refers to when video is displaced horizontally. This may be caused by sync problems.

TelePrompTer - A special monitor mounted in front of a camera so that talent can read text and will appear to be looking at the camera.

thaw - See *unfreeze* .

thumbnail - A small representation of an object. For example, a full screen image might be presented in a much smaller area in an authoring system time line. A *picon* is an example of a thumbnail.

TIFF - Tagged Image File Format time code.

tilt - A camera movement where the camera pivots up or down on its stationary tripod.

timbre - The distinctive tone of a musical instrument or human voice that distinguishes it from other sounds.

time code - See *SMPTE time code* .

time-line processor - A type of authoring facility that displays an event as elements that represent time from the start of the event.

tint - See *hue.*

TMSF - A time format expressed in tracks, minutes, seconds, and frames, which is used primarily by compact disc audio devices.

tone (bass, treble, etc...) - A control that adjusts the various attributes of the audio.

tool palette - A palette containing choices that represent tools, often used in media editors (such as graphics and audio editors). For example, a user might select a "pencil" choice from the tool palette to draw a line in the window.

touch area - (1) An area of a display screen that is activated to accept user input. (2) Synonymous with *anchor, hot spot, and trigger* .

track - A path associated with a single Read/Write head as the data medium moves past it.

track advance - To advance the medium to the beginning of the next track.

track reverse - To rewind the medium to the beginning of the current track. If it is at the beginning of the track it will then jump to the beginning of the previous track.

transform device - A device that modifies a signal or stream received from a transport device. Examples are amplifier-mixer and overlay devices.

translator - A computer program that can translate. In telephone equipment the device that converts dialed digits into call-routine information.

transparency - Refers to when a selected color on a graphics screen is made transparent to allow the video "behind it" to become visible. Often found in dual plane video subsystems.

transparent color - Video information is considered as being present on the video plane which is maintained behind the graphics plane. When an area on the graphics plane is painted with a transparent color, the video information in the video plane is made visible. See also *dual plane video system* .

transport device - A device that plays, records, and positions a media element, and either presents the result directly or sends the material to a *transform device* . Examples are videodisc players, CD-ROMs, and digital audio (wave) player.

treatment - A detailed design document of the video.

tremolo - A vibrating effect of a musical instrument produced by small and rapid amplitude variations to produce special musical effects.

trigger - (1) An area of a display screen that is activated to accept user input. (2) Synonymous with *anchor, hot spot, and touch area* .

truck - In videotaping, a sideways camera movement of the tripod on which the camera is mounted.

tweening - (1) The process of having the computer draw intermediate animation frames between key frames. In other words, the animation tool requires only that pictures of key sections of a motion be provided; the software calculates all the in-between movements. (2) Synonym for *in-betweening* .

U

Ultimatte - The trade name of a very high-quality, special-effects system used for background replacement and image composites.

U-matic - A video cassette system using 0.75-inch tape format.

underrun - Loss of data caused by the inability of a transmitting device or channel to provide data to the communication control logic (SDLC or BSC/SS) at a rate fast enough for the attached data link or loop.

unfreeze - (1) To return to action after a *freeze* . (2) Enables updates to the video buffer. (3) Synonym for *thaw* .

unidirectional microphone - A microphone that responds to sound from only one direction and is not subject to change of direction. (A unidirectional microphone is the type of microphone employed in computers capable of voice recognition.)

unload - To eject the medium from the device.

user-defined message - A private message sent directly to an I/O procedure by using the mmioSendMessage function. All messages begin with an MMIOM prefix, with user-defined messages starting at MMIOM_USER or above.

user interface - The area at which a user and an object come together to interact. As applied to computers, the ensemble of hardware and software that allows a user to interact with a computer.

user's conceptual model - A user's mental model about how things should work. Much of the concepts and expectations that make up the model are derived from the user's experience with real-world objects of similar type, and experience with other computer systems.

V

value set - A control used to present a series of mutually exclusive graphical choices. A tool palette in a paint program can be implemented using a value set.

VCR - Videocassette recorder.

VDD - Virtual device driver.

VDH - Virtual device helper.

VDP - Video display processor.

vector graphics - See *coordinate graphics* .

vendor specific drivers - An extension to an MCD to execute hardware specific commands.

VHS - Very high speed. A consumer and industrial tape format (VHS format).

vicon - A vicon, or video icon, can be an animation or motion video segment in icon size. Usually this would be a short, repeating segment, such as an animation of a cassette tape with turning wheels.

video - Pertaining to the portion of recorded information that can be seen.

video aspect ratio - See *aspect ratio* .

video attribute control - Provides access to and operation of the standard video attributes: brightness, contrast, freeze, hue, saturation, and sharpness. All device communication and user interface support is handled by the control.

video attributes - Refers to the standard video attributes: brightness, contrast, freeze, hue, saturation, and sharpness.

video clip - A section of filmed or videotaped material.

video clipping - See *clipping* .

video digitizer - Any system for converting analog video material to digital representation. (For example, see *DVI* .)

video display buffer - The buffer containing the visual information to be displayed. This buffer is read by the video display controller.

video display controller - The graphics or video adapter that connects to a display and presents visual information.

video encoder - A device (adapter) that transforms the high-resolution digital image from the computer into a standard television signal, thereby allowing the computer to create graphics for use in video production.

video graphics adapter - A graphics controller for color displays. The pel resolution of the video graphics adapter is 4:4.

video image - (1) A still video image that has been captured. (2) Synonymous with *image* and *still image* .

video monitor - A display device capable of accepting a video signal that is not modulated for broadcast either on cable or over the air; in videotaping, a television screen located away from the set where the footage can be viewed as it is being recorded.

video overlay - See *overlay* .

video overlay device - See *overlay device* .

video plane - In a dual plane video system, the video plane contains the video. This video plane will be combined with the graphics plane to create an entire display image.

video programming interface (VPI) - A subsystem that performs output from video source to video window.

video quality - The compression quality level setting to be set for the CODEC. This value is in the range of 0 (min) - 100 (max).

video record rate - Frame rate for recording as an integral number of frames per second. This sets the target capture rate, but there are no assurances this rate will be attained. Drop frame records will be inserted into the output data stream to indicate frames dropped during the capture/record process.

video record frame duration - Frame rate for recording as the time duration of each frame in microseconds. Useful for setting non-integer frame rates, for example, 12.5 FPS of a PAL videodisc: 1000000/12.5 = 8000 microseconds.

video signal - An electrical signal containing video information. The signal must be in some standard format, such as NTSC or PAL.

VSD - Vendor Specific Driver

video scaling - (1) Expanding or reducing video information in size or area. (2) See also *aspect ratio* .

video scan converter - A device that emits a video signal in one standard into another device of different resolution or scan rate.

video segment - A contiguous set of recorded data from a video track. A video segment might or might not be associated with an audio segment.

video signal - An electrical signal containing video information. The signal must be in some standard format, such as NTSC or PAL.

video source selection - The ability of an application to change to an alternate video input using the **connector** command.

video tearing - See *tearing* .

video teleconferencing - A means of telecommunication characterized by audio and video transmission, usually involving several parties. Desktop video teleconferencing could involve having the audio and video processed by the user's computer system, that is, with the other users' voices coming through the computer's speaker, and video windows of the other users displayed on the computer's screen.

videocassette recorder (VCR) - A device for recording or playing back videocassettes.

videodisc - A disc on which programs have been recorded for playback on a computer (or a television set); a recording on a videodisc. The most common format in the United States and Japan is an NTSC signal recorded on the optical reflective format.

videodisc player control - Provides access to and operation of the following videodisc functions: eject, pause, play forward, play reverse, position, record, repeat, rewind, scan forward, scan reverse, step forward, step reverse, and stop. All device communication and user interface support is handled by the control.

videotape - (1) The tape used to record visual images and sound. (2) To make a videotape of. (3) A recording of visual images and sound made on magnetic tape. (All shooting is done in this format, even if the results are later transferred to videodisc or film.)

videotape recorder (VTR) - A device for recording and playing back videotapes. (The professional counterpart of a consumer VCR.)

videotex - A system that provides two-way interactive information services, including the exchange of alphanumeric and graphic information, over common carrier facilities to a mass consumer market using modified TV displays with special decoders and modems.

video windows - Graphical PM-style windows in which video is displayed. Most often associated with the video overlay device.

view - The form in which an object is presented. The four kinds of views are: composed, contents, settings, and help.

viewport - An area on the usable area of the display surface over which the developer has control of the size, location, and scaling, and in which the user can view all or a portion of the data outlined by the window.

virtual device helper - A system service that is available to perform essential functions.

VO - Script abbreviation for *voice-over* .

voice-over - (1) The voice of an unseen narrator in a video presentation. (2) A voice indicating the thoughts of a visible character without the character's lips moving.

volume - The intensity of sound. A volume of 0 is minimum volume. A volume of 100 is maximum volume.

VPI - Video programming interface.

VTR - Videotape recorder.

W

walk-through - A type of animated presentation that simulates a walking tour of a three-dimensional scene.

walk-up-and-use interface - An interface that the target audience should be able to use without having to read manuals or instructions, even if they have never seen the interface.

waveform - (1) A graphic representation of the shape of a wave that indicates its characteristics (such as frequency and amplitude). (2) A digital method of storing and manipulating audio data.

wide shot - Synonym for *long shot* .

wild footage - Synonym for *original footage* .

window - An area of the screen with visible boundaries within which information is displayed. A window can be smaller than or the same size as the screen. Windows can appear to overlap on the screen.

window coordinates - The size and location of a window.

wipe - Technical effect of fading away one screen to reveal another.

workplace - A container that fills the entire screen and holds all of the objects that make up the user interface.

write once/read many (WORM) - Describes an optical disc that once written to, cannot be overwritten. Storage capacity ranges from 400MB to 3.2GB. Present technology allows only one side to be read at a time; to access the other side, the disk must be turned over.

WS - Script abbreviation for *wide shot* .

WYSIWYG - What You See Is What You Get. The appearance of the object is in actual form. For example, a document that looks the same on a display screen as it does when it is printed. Composed views of objects are often WYSIWYG.

X Y Z

Y - Refers to the luminance portion of a Y/C video signal.

Y/C - Color image encoding that separates luminance ((Y) and *chrominance* (C) signals.

YIQ - Image encoding scheme similar to YUV that selects the direction of the two color axes, I and Q, to align with natural images. As an average, the I signal bears much more information than the Q signal. (YIQ is used in the NTSC video standard.)

YUV - Color image encoding scheme that separates luminance (Y) and two color signals: red minus Y (U), and blue minus Y (V). Transmission of YUV can take advantage of the eye's greater sensitivity to luminance detail than color detail.

zoom in - An optical camera change where the camera appears to approach the subject it is shooting.

zooming - The progressive scaling of an image in order to give the visual impression of movement of all or part of a display group toward or away from an observer.

zoom out - An optical camera change where the camera appears to back up from the subject it is shooting.
